

Projet de programmation: Racetrack

L'objectif de ce projet est d'implémenter un petit jeu graphique de type “puzzle” et un algorithme de recherche automatique de solution.

Le jeu *Racetrack*

Racetrack (aussi connu sous les noms *Vector Race*, *Graph Racers*, *Pencil and paper racing*, et bien d'autres !) est un jeu de course automobile très simple, jouable avec un crayon sur une simple feuille de papier quadrillée. Le jeu consiste à tracer la trajectoire la plus rapide possible, depuis la zone de départ jusqu'à la zone d'arrivée. À chaque tour, le joueur prolonge sa trajectoire actuelle en marquant une nouvelle position pour sa voiture. La nouvelle position doit respecter certaines règles simulant l'accélération et l'inertie de la voiture.

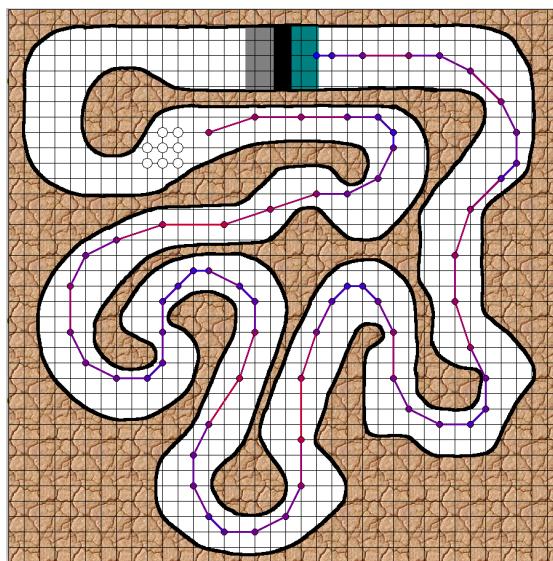


Fig. 1 : Une partie de *Racetrack* en cours.

Dans la partie ci-dessus, les cercles colorés indiquent les positions successives de la voiture, reliées par des lignes colorées en fonction de la vitesse, et les cercles blancs indiquent les positions possibles pour le prochain coup.

Règles du jeu

- Le jeu se joue sur une piste rectangulaire quadrillée. Chaque intersection représente une position, qui peut être soit la zone de départ (en bleu-vert sur l'image ci-dessus), soit la zone d'arrivée (en gris), soit une partie de la piste (en blanc), soit un obstacle hors-piste (tout le reste).

- La trajectoire de la voiture est marquée par une ligne brisée représentant toutes les positions successives de la voiture. Chaque position est une intersection sur la grille.
- Au premier tour, le joueur choisit une position initiale parmi les positions se trouvant dans la zone de départ.
- À chaque tour suivant, le joueur prolonge la trajectoire actuelle en marquant une nouvelle position. Afin de simuler l'inertie de la voiture, la nouvelle position doit respecter les règles suivantes :
 - Le *vecteur* porté par le dernier segment de la trajectoire représente la vitesse actuelle de la voiture.
 - Par défaut, au coup suivant, la voiture conserve sa vitesse. C'est-à-dire que la position suivante de la voiture sera la position obtenue en appliquant le vecteur calculé à la position actuelle. Cette position est appelée le *point principal*.
 - Le joueur peut décider de se rendre au point principal, ou bien faire varier la vitesse (intensité et / ou direction) de la voiture en choisissant à la place l'un des 8 voisins du point principal.

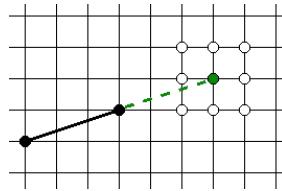


Fig. 2 : Le point principal (vert) et les 8 autres coups autorisés.

- La voiture ne doit pas rencontrer un obstacle le long du chemin.
- Si la trajectoire s'arrête dans la zone d'arrivée, la partie est gagnée et on compte le nombre de coups joués pour y parvenir. Le but est de réaliser la trajectoire la plus rapide, c'est-à-dire de gagner avec le moins de coups possibles.
- S'il arrive que le joueur n'ait aucun coup valide, par exemple parce que les 9 coups possibles pour prolonger la trajectoire sortent de la piste ou rencontrent un obstacle, la partie est perdue.

Réalisation

Le projet est constitué de quatre tâches principales. Chaque tâche contient plusieurs niveaux de perfectionnement. Vous devez avoir réalisé au moins le niveau 1 de toutes les tâches obligatoires avant d'aborder les améliorations facultatives. Les niveaux au dessus du niveau 1 sont volontairement moins guidés : à vous de concevoir une solution adaptée !

Tâche 1 : Moteur du 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 au cours de la partie.

Représentation de l'état du jeu

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

La **piste** est représentée au choix par une liste de chaînes de caractères, ou une liste de listes de caractères. Les dimensions des listes et des chaînes correspondent aux dimensions de la piste représentée. Ainsi, la valeur `piste[i][j]` décrit le type de la zone située à l'intersection de la i-ème ligne et de la j-ème colonne, avec des caractères désignés pour représenter les zones de route (.), départ (>), arrivée (*) et obstacle (#).

La **trajectoire** est représentée par une liste de couples d'entiers, représentant chacune une position de la voiture. La position actuelle de la voiture est donnée par la *dernière* valeur de la liste.

Par exemple, la partie ci-dessous est représentée par les structures suivantes :

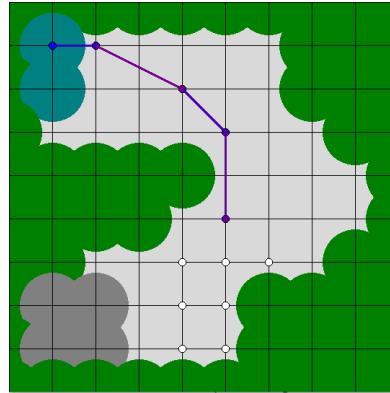


Fig. 3 : Une partie sur un tout petit circuit.

```

piste_chaine = ['# ##### #####',      piste_liste = [[ '#', '#', '#', '#', '#', '#', '#', '#', '#', '#'],
               '>....###',          '#', '>', '.', '.', '.', '.', '#', '#', '#'],
               '>....###',          '#', '>', '.', '.', '.', '.', '#', '#', '#'],
               '#.....##',          '#', '.', '.', '.', '.', '.', '.', '#', '#'],
               '####...##',          '#', '#', '#', '#', '.', '.', '.', '.', '#'],
               '####...##',          '#', '#', '#', '#', '.', '.', '.', '.', '#'],
               '#....##',           '#', '#', '.', '.', '.', '.', '.', '#', '#'],
               '**...####',          '#', '*', '*', '.', '.', '#', '#', '#', '#'],
               '**...####',          '#', '*', '*', '.', '.', '#', '#', '#', '#'],
               '##### #####']        '#', '#', '#', '#', '#', '#', '#', '#', '#', '#']]
```

```

trajectoire = [ (1,1), (1,2), (2,4), (3,5), (5,5) ]
```

Déplacements de la voiture

Le moteur du jeu doit permettre de calculer les déplacements autorisés à chaque tour du jeu. Pour ce faire, on propose d'implémenter au minimum les fonctions suivantes :

- `vitesse(trajectoire)` : renvoie la vitesse actuelle de la voiture, sous la forme d'une paire de coordonnées.
- `verif_collision(debut, fin)` : vérifie que le segment (`debut, fin`) n'entre pas en collision avec un obstacle, où `debut` et `fin` sont chacun des couples d'entiers représentant des positions. Renvoie `True` si le déplacement est autorisé, et `False` sinon.
 - **Niveau 1 : règles souples** : on vérifiera simplement que `debut` et `fin` sont des positions autorisées. Le joueur peut traverser les obstacles tant qu'il ne s'y arrête pas.
 - **Niveau 2 : règles strictes** : le segment ne doit pas croiser d'obstacle, même en dehors des intersections de la grille. Par exemple, le dernier segment de la trajectoire de gauche ci-dessous est autorisé par les règles souples, mais est **interdit** par les règles strictes. La trajectoire de droite est autorisée par les deux règles.

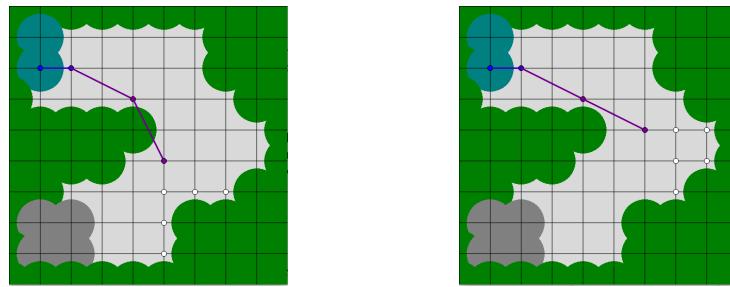


Fig. 4 : La trajectoire de gauche “mord” hors de la piste, celle de droite non.

- `options(trajectoire)` : calcule les coups valides pour prolonger la trajectoire donnée, et renvoie le résultat sous la forme d'une liste de positions autorisées. Utilise la fonction `vitesse` pour calculer le point principal et la fonction `verif_collision` pour déterminer quelles positions parmi les 9 positions autour du point principal sont autorisées.

Si `trajectoire` est vide, la fonction `options` renvoie la liste des positions dans la zone de départ.

Tâche 2 : Représentation et chargement des pistes

Le but de cette tâche est de permettre au programme de lire et de charger des pistes prédéfinies et stockées dans des fichiers.

- **Niveau 1 : fichier texte.** La piste est représentée par un fichier texte composé des caractères ., #, > et *, représentant, ligne par ligne, le contenu de la piste.

Un exemple de fichier avec la piste correspondante est donné ci-dessous :

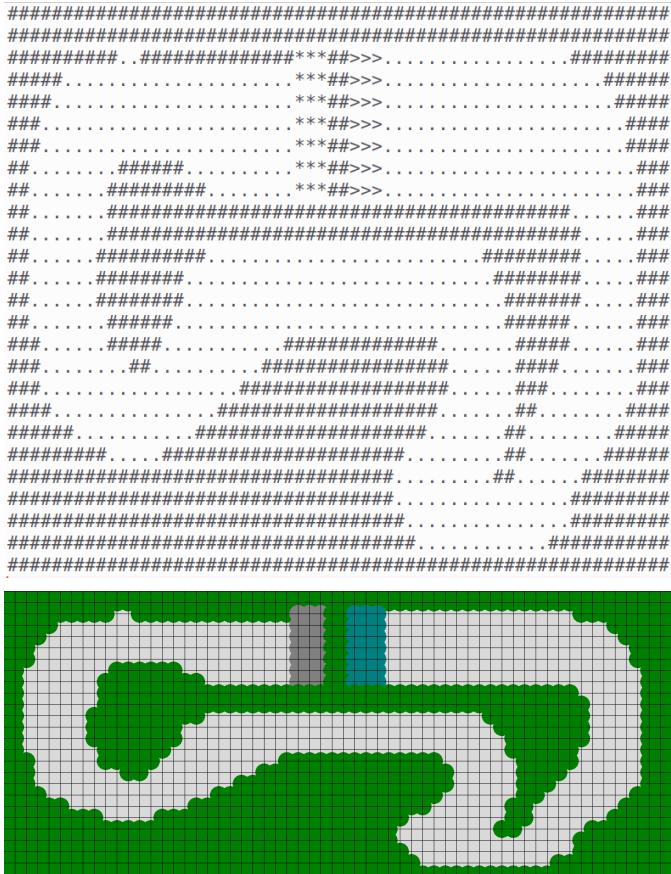


Fig. 4 : Le fichier d'une piste, et la piste correspondante.

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.

- **Niveau 2 : fichier image.** Pour obtenir un visuel plus attrayant, la piste est cette fois-ci représentée par une image, par exemple au format png. L'image respecte un code couleur pré-établi : blanc pour une case de piste, bleu-vert pour la zone de départ, et gris pour la zone d'arrivée. Tous les autres éléments représentent des obstacles. Un exemple d'une telle image est donné ci-dessous :

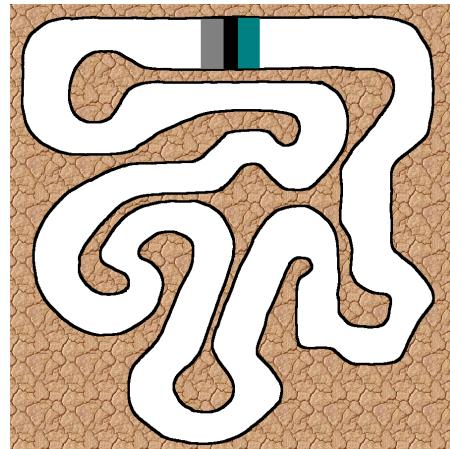


Fig. 5 : Une image simplifiée représentant une piste.

Cette amélioration nécessite entre autres de revoir les structures de données utilisées à la tâche 1. De plus, la grille n'est pas dessinée sur l'image. Votre programme devra choisir un maillage d'une taille appropriée à l'image, ou bien permettre au joueur de choisir la taille du maillage.

Indication : vous pouvez utiliser `piste_img = fltk.PhotoImage(file = chemin/d'accès/de/l'image)` pour obtenir dans la variable `piste_img` une représentation Python du fichier image. Vous pouvez ensuite utiliser `piste_img.get(x,y)` pour obtenir la couleur du pixel aux coordonnées `(x,y)`.

- **Niveau 3 : décors.** Finalement, on souhaite pouvoir décorer la piste avec des éléments qui ne sont pas considérés comme des obstacles. De plus, on ne veut plus utiliser le code couleur, qui nuit à l'esthétique du jeu. Réfléchissez à une méthode pour décorrérer le visuel de la logique interne du jeu. Par exemple, on veut pouvoir jouer sur des images plus complexes, comme celle ci-dessous.



Fig. 6 : Une image complexe représentant une piste. (Source : freepik)

Indication : si vous en êtes là, vous savez ce que vous faites ! Voici tout de même un petit indice : utilisez *deux* images !

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 une piste de son choix parmi un ensemble de pistes disponibles.

Une fois le jeu lancé, la piste et le quadrillage devront être affichés graphiquement. Le visuel dépendra du niveau de perfectionnement choisi lors de la tâche 2. À chaque tour, le joueur pourra jouer en cliquant sur les différentes options de déplacement qui lui seront proposées, et le jeu se chargera de tracer l'historique des positions occupées par la voiture.

Le joueur pourra choisir d'annuler le dernier coup joué avec la touche **Backspace** (retour arrière), et revenir au menu pour choisir une autre piste avec la touche **Echap**. En cas de victoire, un message approprié doit s'afficher et proposer au joueur de revenir au menu.

Niveau 2 : représentation de la vitesse. la couleur de chaque segment de la trajectoire doit dépendre de la longueur du segment, pour indiquer visuellement les sections où la voiture a ralenti ou accéléré. Un exemple est donné sur la Figure 1.

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

Tâche 4 : Recherche automatique de solution

La quatrième tâche du projet consiste à implémenter un algorithme de recherche automatique de solution, ou *solveur*, pour le jeu Racetrack. Le rôle du solveur est de déterminer automatiquement une trajectoire respectant les règles du jeu, depuis la zone de départ jusqu'à la zone d'arrivée.

Il existe plusieurs approches pour construire un tel solveur. Chaque approche a ses avantages et ses défauts, en terme de rapidité d'exécution ou de qualité de la trajectoire obtenue.

Quel que soit le niveau implémenté, votre algorithme devra permettre de visualiser sur l'interface graphique les trajectoires explorées au cours de la recherche et afficher la trajectoire gagnante qui a été trouvée.

Niveau 1 : recherche en profondeur

Nous allons commencer par implémenter un algorithme de recherche naïf (appelé algorithme de recherche *par backtracking*, ou algorithme de *recherche en profondeur*) qui va essayer de prolonger une trajectoire donnée en essayant tous les coups autorisés, puis rechercher récursivement à prolonger les nouvelles trajectoires ainsi obtenues.

Pour garantir que l'algorithme ne provoque pas d'appels récursifs infinis, nous allons nous assurer qu'il ne repasse jamais deux fois par la même position avec la même vitesse. Un couple (**position**, **vitesse**) est appelée une *configuration* du jeu. Nous allons donc enregistrer chaque configuration visitée dans un ensemble **visite** de type `set` et interrompre prématurément les appels récursifs qui atteignent des configurations déjà visitées.

L'algorithme procède comme suit, à partir d'une trajectoire **trajectoire** et d'un ensemble **visite**, tous deux initialement vides :

1. Si **trajectoire** termine dans la zone d'arrivée, on renvoie `True` : on a bien trouvé une trajectoire depuis la zone de départ jusqu'à la zone d'arrivée.
2. On calcule la configuration `c = (position, vitesse)` du dernier élément de **trajectoire**.

3. Si c est déjà dans `visite`, on renvoie `False`. Cette configuration a déjà été considérée, on interrompt donc la recherche. Sinon, on ajoute c à `visite`.
4. On parcourt de manière itérative chaque option o dans `options(trajectoire)`:
 - On ajoute o à `trajectoire` pour former une nouvelle trajectoire.
 - On relance récursivement la recherche en profondeur depuis cette nouvelle `trajectoire`.
 - Si la recherche aboutit, on renvoie `True`. Sinon, on retire o de `trajectoire` et on réitère avec l'option suivante o , si elle existe.
5. Si aucune option n'aboutit à une trajectoire, on renvoie `False` : la trajectoire donnée ne peut pas être prolongée en une trajectoire gagnante.

À la fin de l'exécution, soit l'algorithme renvoie `True`, et `trajectoire` aura été rempli avec une trajectoire gagnante, soit il n'existe pas de trajectoire gagnante, l'algorithme renvoie `False`, et `trajectoire` sera vide.

Niveau 2 : recherche en largeur

L'algorithme proposé au niveau 1 a la qualité de toujours trouver une solution, pourvu qu'il en existe au moins une. Cependant, la solution trouvée peut être très mauvaise, au sens où elle nécessitera souvent beaucoup plus d'étapes que la trajectoire optimale. L'exemple ci-dessous montre la trajectoire trouvée par la recherche en profondeur d'une part, et la trajectoire optimale d'autre part.

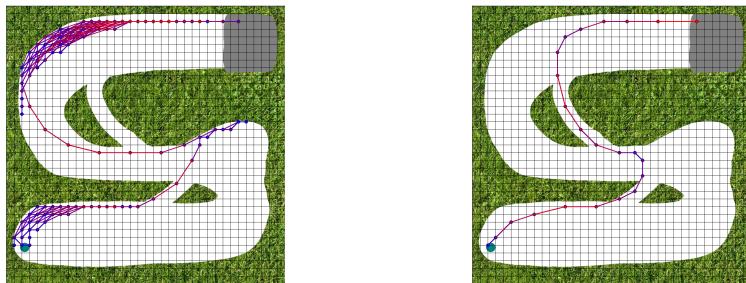


Fig. 7 : Recherche en profondeur (gauche) et trajectoire optimale (droite).

Proposez un algorithme pour trouver la trajectoire optimale !

Indication : utilisez une file !

Niveau 3 : autres stratégies de recherche

L'algorithme du niveau 2 présente deux défauts. Premièrement, la recherche peut prendre beaucoup de temps sur une grande grille. Deuxièmement, la solution trouvée est un peu **trop** bonne : il ne sera pas amusant pour un joueur humain de s'y confronter, puisqu'elle est, par définition, imbattable.

Pour ce niveau, vous devez proposer une ou plusieurs autres stratégies de recherche qui produisent des trajectoires intéressantes : ni trop naïves, ni trop bonnes, de sorte qu'il soit intéressant

pour un joueur humain d'essayer de faire mieux. Votre rapport devra précisément expliquer la démarche que vous avez suivie, et les idées qui font fonctionner votre algorithme.

Vous pouvez bien entendu vous inspirer des stratégies qui ont été discutées en cours ! À titre d'exemple, voici le résultat de deux stratégies implémentées par votre enseignant. Il est facile de constater que ces deux trajectoires **ne sont pas** optimales, mais produisent tout de même un résultat crédible.

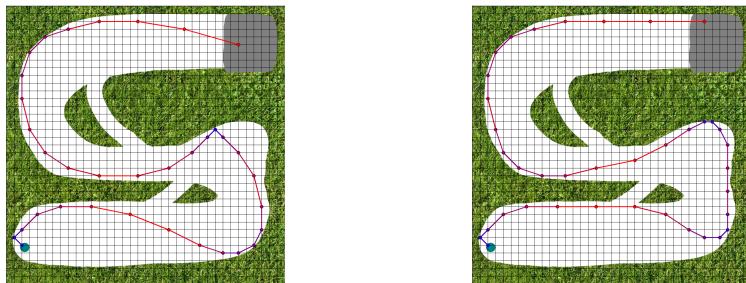


Fig. 8 : Les résultats de deux stratégies de recherche intéressantes.

Tâches complémentaires

Cette section propose des améliorations du projet, à aborder uniquement si toutes les tâches obligatoires ont été terminées **au moins au niveau 1**. Les étoiles (*) devant chaque tâche donnent une indication sur sa difficulté.

- (*) Ajouter une liste de meilleures scores sur chaque piste. À la fin d'une partie, si le joueur a obtenu le meilleur score, l'interface lui proposera d'ajouter son nom à la liste. La liste doit être conservée même lorsque le programme est fermé.
- (*) Implémenter un mode “deux joueurs”. Les joueurs jouent à tour de rôle, et les deux trajectoires sont maintenues et dessinées. Le premier joueur à atteindre la zone d'arrivée gagne la partie. Vous devrez décider d'une pénalité appropriée lorsqu'un joueur ne peut pas jouer.
- (*) Implémenter un mode “fantôme”. À la fin de la partie, le joueur peut recommencer pour essayer d'obtenir une meilleure trajectoire. La meilleure trajectoire obtenue lors des essais successifs reste dessinée en transparence, pour aider le joueur à visualiser les sections qui peuvent être améliorées.
- (**) Implémenter un mode “blitz”. À chaque tour, le joueur dispose de 3 secondes pour décider son prochain déplacement. S'il ne décide pas dans le temps imparti, la voiture continue sur sa trajectoire actuelle. Vous devrez décider d'une pénalité appropriée en cas de crash.
- (**) Ajouter des terrains spéciaux sur la piste qui modifient le comportement de la voiture lorsqu'ils sont traversés. Par exemple, une plaque de verglas empêcherait la voiture de ralentir, un tremplin forcerait la voiture à conserver sa trajectoire, une route de terre ferait ralentir la voiture à chaque tour. **Cette amélioration nécessite d'avoir implémenté les règles strictes.**

- (**) Ajouter un éditeur de pistes, c'est-à-dire une interface graphique permettant à l'utilisateur de dessiner lui-même la piste, en plaçant les obstacles, la zone de départ et la zone d'arrivée. La piste ainsi obtenue sera enregistrée dans un fichier au bon format pour pouvoir être jouée.
- (**) Ajouter des éléments mobiles sur la piste. Par exemple, des barrières s'ouvrent et se ferment, des ponts s'abaissent ou se lèvent, un train suit un parcours le long d'un rail qui croise la piste, etc. **Cette amélioration nécessite de proposer des pistes adaptées, une manière de représenter les éléments mobiles et leurs règles de déplacement, et d'adapter les solveurs pour prendre en compte la dynamique de la piste.**
- (**) Implémenter un générateur de pistes aléatoires. Les pistes générées doivent être *intéressantes* : la zone d'arrivée ne doit pas être trop proche de la zone de départ, la piste doit avoir au moins une solution, la trajectoire optimale ne doit pas être trop simple, les obstacles doivent être placés de façon naturelles, la piste ne doit pas avoir (trop) de sections inutiles, etc. **Vous expliquerez en détail dans votre rapport votre démarche et la manière dont vous avez résolu les différentes difficultés.**

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