Projet de programmation : Push it down !

Objectif : L'objectif de ce projet est de réaliser un petit jeu de type puzzle, où une boule doit atteindre la sortie d'un niveau constitué d'un empilement de blocs. La boule peut se déplacer dans quatre directions, et pousser un bloc qui se trouve devant elle si ce bloc n'en porte pas un autre, et si la case qui se trouve derrière est libre.

Attention : Le sujet est long, mais il est fait pour vous aider. Donc, oui, désolés, **il faut tout lire !**

1 Travail demandé et modalités d'évaluation

1.1 Organisation du travail

- Le projet est **obligatoire** et sa note vaudra pour un tiers (1/3) de la note finale d'AP2.
- Le projet est à réaliser en **binômes** (deux étudiants au moins, deux étudiants au plus !).
- Le travail fera (probablement) l'objet d'une soutenance. Les dates de rendu et de soutenances seront précisées en temps voulu.
- Les tâches pourront être réparties au sein du binôme, ou l'ensemble du projet réalisé conjointement. Chaque membre du binôme est censé pouvoir expliquer l'ensemble du code.
- Il est encouragé de demander de l'aide à vos chargés de TP, de TD, de cours. Vous avez le droit d'échanger des **idées** avec d'autres étudiants (par exemple sur le forum). Cependant, **tout plagiat détecté entre des binômes différents sera lourdement sanctionné** (on entend par plagiat la recopie de portions non négligeables de code).
- Il est possible d'obtenir une bonne note sans traiter la moindre amélioration optionnelle, si la base du projet est bien réalisée et l'ensemble des consignes respectées.
- En cas de difficulté d'organisation **importante** (problème de binôme, problème technique, etc.), mettez un mot sur le forum ou contactez rapidement votre chargé de TP, et à défaut l'un des autres enseignants du module.

1.2 Description des fichiers à rendre

Le rendu du projet devra être constitué d'une **unique** archive, **impérativement au format zip**. Les correcteurs se réservent le droit de ne pas décompresser toute archive d'un autre type (RAR ou autre). Cette archive devra contenir :

- Un fichier `pushit.py` contenant le programme principal du jeu, ainsi que tout autre fichier Python nécessaire à son bon fonctionnement. Le lancement du jeu devra impérativement se faire par la commande `python3 pushit.py`.
- Un sous-répertoire `maps/` contenant tous les niveaux du jeu que vous aurez réalisés, chacun de ces niveaux respectant le format donné ci- dessous. Vous pouvez si vous le souhaitez répartir ces niveaux au sein de plusieurs sous-répertoires du répertoire `maps/`.
- Un fichier `rapport.pdf` **ou** `rapport.html` contenant le compte-rendu de votre travail ainsi que la documentation de votre jeu.

Le rendu du projet doit être fait sur *elearning* par **un seul** des deux membres du binôme. **Conseil :** N'abordez les paragraphes « améliorations » qu'après avoir **entièrement** terminé toutes les questions obligatoires des **deux** parties du projet.

1.3 Contenu du rapport

Le rapport détaillera de façon **précise** et **succincte** (maximum 5 pages) :

- Le nom et le groupe de TP des deux étudiants ayant participé au projet.
- Un guide utilisateur **bref** indiquant comment se servir de chaque programme, des menus, d'options éventuelles en ligne de commande.
- L'état d'avancement des parties obligatoires du projet (ce qui est terminé et ce qui ne l'est pas), les bugs éventuels.
- Une description des améliorations éventuelles.
- Pour **chaque fonction réalisée**, des commentaires sur son rôle, son implémentation, etc.
- La répartition du travail entre les membres du binôme, les difficultés rencontrées.

1.3.1 Barème indicatif

Voici comment se décomposera à peu près la note finale (ce barème est susceptible de changer légèrement) :

- **8 pts.** Fonctionnalités de base : jeu utilisable, quelques niveaux réalisés, blocs déplaçables, pas de bugs de déplacement, possibilité de recommencer un niveau et de passer au niveau suivant, annulation des derniers coups, solveur simple.
- **2 pts.** Qualité du code : clarté, structuration en fonctions, commentaires).
- **3 pts.** Respect des consignes de rendu (format et contenu de l'archive, noms de fichiers, etc.) et qualité du rapport.
- **3 pts.** Qualité de la soutenance. **Attention,** la soutenance est commune mais chaque membre du binôme peut recevoir une note différente. D'autre part, l'absence à la soutenance ou un évident manque de compréhension du code donnera lieu à un malus important sur la note globale.
- **4 pts ou plus :** Améliorations éventuelles : ergonomie et apparence du jeu, fonctionnalités supplémentaires, etc. Voir section 7 pour une liste de suggestions.

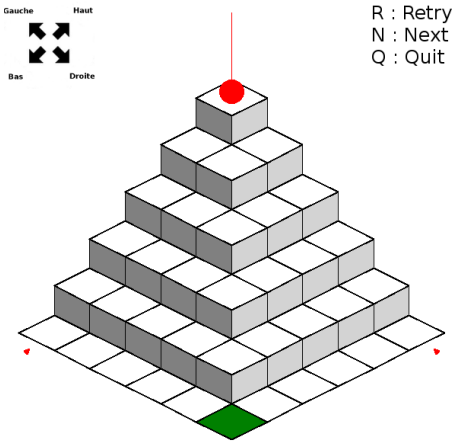
Enfin, consigne la plus importante : **amusez-vous**, soyez créatifs et apprenez à devenir de meilleurs programmeurs !

2 Encodage des niveaux

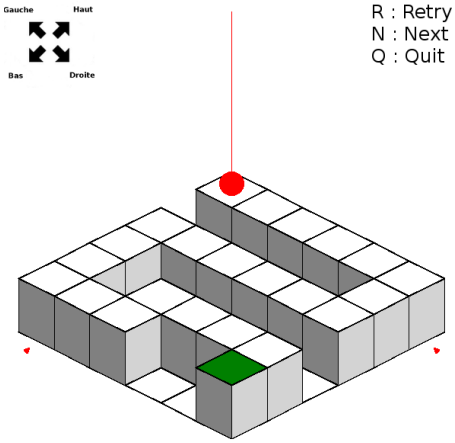
Un niveau du jeu est constitué d'un empilement de blocs identiques sur un plateau carré de dimensions n par n. Chaque case du plateau peut supporter 0, 1 ou plusieurs blocs. Un tel niveau est donc simplement représenté par une matrice carrée (autrement dit une liste de listes) d'entiers positifs ou nuls, chacun indiquant le nombre de blocs empilés sur la case correspondante. Si `niv` est une telle matrice, `niv[0][0]` représente le nombre de blocs empilés dans le coin supérieur gauche du plateau, et `niv[i][j]` la hauteur de blocs sur la `j`-ème case de la `i`-ème colonne.

Les niveaux peuvent être de grande taille, mais doivent rester jouables. La base sera toujours un carré, et la position de départ sera toujours `(0, 0)`, c'est à dire le coin supérieur gauche de la matrice d'entiers. La sortie du niveau sera toujours la case opposée, c'est à dire la case se trouvant à la position `(n-1, n-1)` pour un niveau de taille n. La taille minimale d'un niveau est 2 pour que les positions de départ et d'arrivée soient différentes (sinon le joueur à gagné avant même de commencer à jouer).

Voici deux exemples de niveaux tels qu'ils sont dessinés par notre prototype du jeu, et leur représentation matricielle :




```
[[5, 4, 3, 2, 1, 0],
 [4, 4, 3, 2, 1, 0],
 [3, 3, 3, 2, 1, 0],
 [2, 2, 2, 2, 1, 0],
 [1, 1, 1, 1, 1, 0],
 [0, 0, 0, 0, 0, 0]]
```




```
[[1, 1, 1, 1, 1, 1],
 [0, 0, 0, 0, 0, 1],
 [1, 1, 1, 1, 1, 1],
 [1, 0, 0, 0, 0, 0],
 [1, 0, 1, 1, 1, 1],
 [1, 1, 1, 0, 0, 1]]
```

Pour plus de commodité on stockera les niveaux dans des fichiers textes contenant les valeurs de la matrice. Le format est très simple : chaque ligne du fichier représente une ligne de la matrice sous la forme d'une suite d'entiers séparés par des espaces, sans crochets ni les virgules.

Le niveau de gauche sur l'image ci-dessus, en forme de tour, pourra ainsi être stocké dans le fichier `maps/map_tour.txt` suivant :

```
~/python/push_it$ cat maps/map_tour.txt
5 4 3 2 1 0
4 4 3 2 1 0
3 3 3 2 1 0
2 2 2 2 1 0
1 1 1 1 1 0
0 0 0 0 0 0
```

Le second niveau, en forme de `2` sera :

```
~/python/push_it$ cat maps/map_2.txt
1 1 1 1 1 1
0 0 0 0 0 1
1 1 1 1 1 1
1 0 0 0 0 0
1 0 1 1 1 1
1 1 1 0 0 1
```

3 Dessin du plateau de jeu

Votre projet devra manipuler par défaut une fenêtre de taille 600 par 600 pixels. Ce n'est pas très grand mais cette taille reste praticable pour la quasi-totalité des ordinateurs. Voici quelques indications pour réussir l'affichage du jeu en perspective cavalière (aussi appelée 3D isométrique).

3.1 Dessiner un bloc à la bonne échelle

Pour dessiner un bloc en perspective cavalière, il faut s'y prendre correctement. Dans l'objectif de vous épargner les calculs géométriques les plus fastidieux, nous vous fournissons quelques fonctions pour afficher les blocs (originellement en 3d) sur un écran d'ordinateur (qui est en 2d). L'extrait de code fourni est reproduit en figure 1. Prenez le temps de bien l'analyser pour le comprendre. Si par hasard vous souhaitiez personnaliser votre interface graphique (notamment avec des couleurs), il vous sera nécessaire de bien identifier où doivent s'opérer les modifications.

Pour mener à bien le rendu graphique et réaliser ces fonctions, vos enseignants ont d'abord fait un dessin. L'objectif est de prendre en charge toutes tailles de grilles en utilisant au maximum les 600×600 pixels de la fenêtre graphique. La figure 2 montre le croquis. Un choix possible pour les dimensions *lb* et *hb* des blocs est :

$$lb = 280/\textit{taille de la grille}$$

$$hb = min(1.5 \times lb, 230/(\textit{hauteur max} + 1))$$

la *taille de la grille* étant le nombre de cases par ligne (et aussi le nombre de lignes) et *hauteur max* étant le nombre maximum de blocs empilés sur une case de la grille. L'utilisation d'un calcul de minimum évite ici d'avoir des blocs trop allongés en hauteur qui donneraient un rendu graphique moins agréable.

```python
def coin_bas(i, j, k, lb, hb):
    """
    Calcule les coordonnées, en pixels, du coin le plus bas d'un bloc représenté par le triplet
    (i, j, k), où i est le numéro de ligne et j le numéro de colonne de la case sur laquelle est
    posé le bloc, et k est sa hauteur. Reçoit aussi les dimensions lb et hb d'un bloc.
    """
    x = 300 + (j-i) * lb
    y = 300 + (j+i) * lb//2 - (k-1) * hb + lb
    return x, y

def affiche_bloc(i, j, k, lb, hb, c="white"):
    """
    Affiche le bloc de coordonnées (i, j, k) conformément au schéma donné dans le sujet. Reçoit
    aussi les dimensions lb et hb d'un bloc ainsi qu'un paramètre optionnel c indiquant la couleur
    de la face supérieure du bloc.
    """
    # calcul des coordonnées du coin bas du bloc
    x, y = coin_bas(i, j, k, lb, hb)
    # calcul des coordonnées des autres sommets inférieurs du bloc
    xg, xd, ymb = x - lb, x + lb, y - lb//2
    # calcul des ordonnées des sommets supérieurs
    ybh, ymh, yhh = y - hb, y - lb//2 - hb, y - lb - hb
    # dessin de la face supérieure, en vert si c'est l'arrivée
    face_haut = [(x, ybh), (xd, ymh), (x,  yhh), (xg, ymh)]
    polygone(face_haut, remplissage=c, epaisseur=2)
    # dessin des faces latérales si hauteur non nulle
    if k > 0:
        face_gauche = [(x, y),   (xg, ymb), (xg, ymh), (x,  ybh)]
        face_droite = [(x, y),   (xd, ymb), (xd, ymh), (x,  ybh)]
        polygone(face_gauche, remplissage='grey')
        polygone(face_droite, remplissage='lightgrey')

def affiche_bille(i, j, k, lb, hb, n):
    """
    Affiche la bille aux coordonnées (i, j, k). Reçoit aussi les dimensions lb et hb d'un bloc
    et la taille n du plateau.
    """
    # dessin de la bille proprement dite
    x, y = coin_bas(i, j, k, lb, hb)
    cercle(x, y - 2*lb//3, lb//3, couleur="red", remplissage="red")
    # repère vertical pour une meilleure visibilité
    ligne(x, y - 2*lb//3, x, 20, couleur='red')
    # flèche-repère de gauche
    x, y = coin_bas(n-1, j-0.5, 1, lb, hb)
    fleche(x - 20, y + 20, x - 10, y + 10, couleur="red", epaisseur=3)
    # flèche-repère  de droite
    x, y = coin_bas(i-0.5, n-1, 1, lb, hb)
    fleche(x + 20, y + 20, x + 10, y + 10, couleur="red", epaisseur=3)

if __name__ == "__main__":
    # exemple de programme (à modifier évidemment...)
    cree_fenetre(600, 600)
    affiche_bloc(0, 0, 2, 50, 50)
    affiche_bloc(0, 1, 1, 50, 50)
    affiche_bloc(1, 0, 1, 50, 50)
    affiche_bloc(1, 1, 0, 50, 50, c="green")
    affiche_bille(0, 0, 3, 50, 50, 2)
    attente_clic()
```

FIG. 1: Code fourni (fichier `pushit.py`), à compléter.

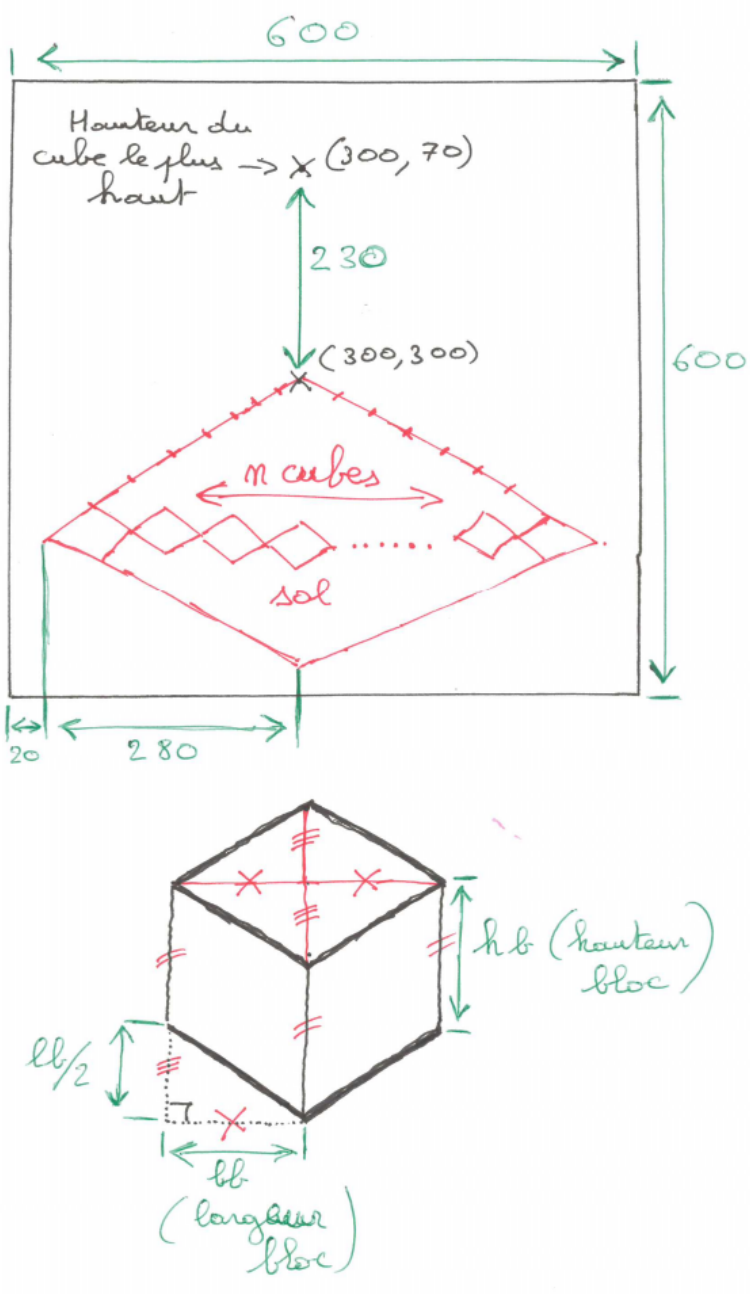


FIG. 2: Découpage de la fenêtre graphique pour y placer la grille.

3.2 Dessiner un niveau complet

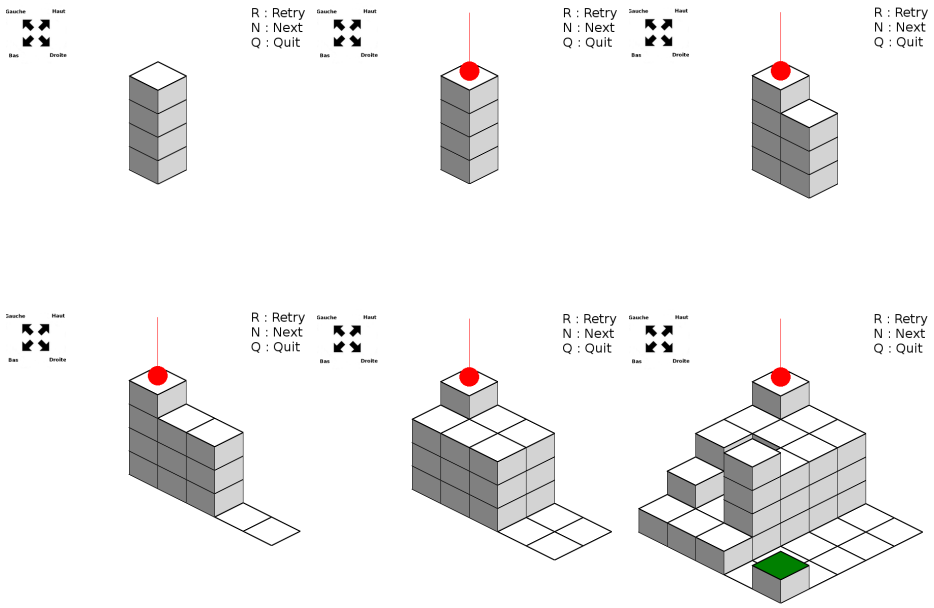
Lorsque l'on sait dessiner un bloc, pour dessiner un niveau il suffit de dessiner tous les blocs, à condition de le faire dans le bon ordre ! En effet, pour obtenir un rendu correct il faut que les blocs les plus hauts cachent les blocs les plus bas, et les blocs les plus "proches" cachent les blocs les plus "lointains".

Une manière de procéder pour arriver à ce résultat consiste à travailler ligne par ligne et de gauche à droite, en partant de la case de coordonnées $(0\,,0)$, puis $(0\,,1)$, etc., et de dessiner pour chaque case $(i\,,j)$ les blocs de bas en haut. Si la bille se trouve à la position $(i\,,j)$, on la dessine après avoir dessiné tous les blocs de cette case.

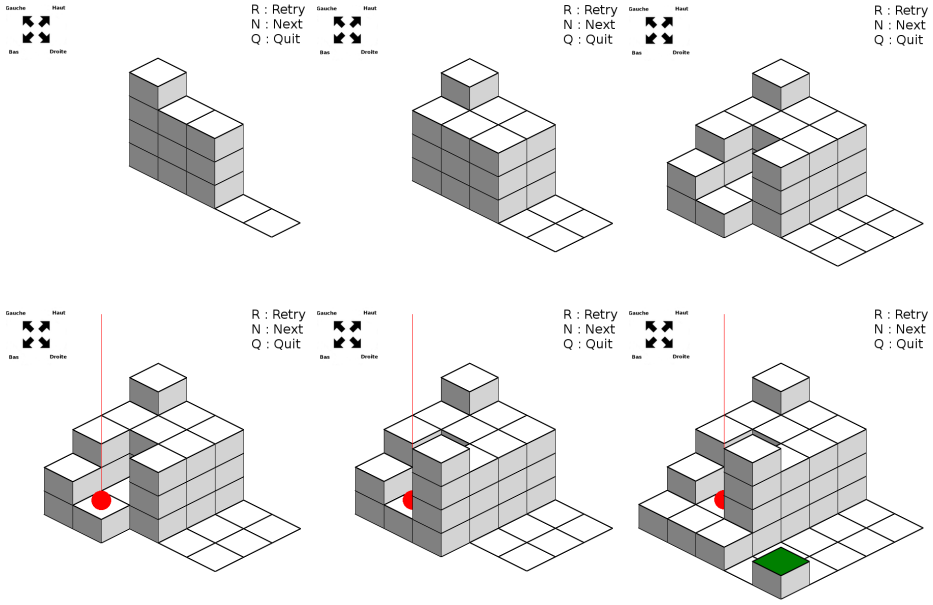
Les images ci-dessous montrent comment procéder. Considérons le plateau associé au fichier suivant :

```
~/python/push_it$ cat maps/map6.txt
4 3 3 0 0
3 3 3 0 0
3 1 3 0 0
2 1 4 0 0
1 1 1 0 1
```

En s'arrêtant après chaque case de la première ligne, puis encore après la seconde ligne, on obtiendrait les images suivantes :



Ici la bille est dessinée après tous les blocs de position $(0\,,0)$ car elle se trouve à la position de départ. Au fur et à mesure que je joueur déplace la bille, elle sera dessinée à un autre moment. Voici quelques étapes de dessin du même niveau si la bille se trouve en position $(3\,,1)$:



Vous remarquerez qu'il a été rajouté des repères visuels pour que le joueur arrive à situer la bille quand cette dernière est partiellement cachée par d'autre blocs. Ce n'est pas obligatoire mais peut aider à la jouabilité.

4 Déplacements et autres commandes

Vous devez gérer deux types de déplacements : ceux qui laissent la grille inchangée (seule la bille bouge) et les déplacements affectant la grille car un bloc est poussé.

Quelques autres commandes liées à d'autres touches du clavier doivent être disponibles à tout moment, elles sont détaillées au paragraphe 4.3.

4.1 Les déplacements standards

La bille peut aller dans les quatre directions sur les cases voisines de la case où elle est située. Il faut d'une part que ces cases existent et ce n'est toujours le cas sur les bords. D'autre part, la bille ne peut que descendre. Sur la grille suivante, la bille peut atteindre n'importe quelle case à partir de n'importe quelle case :

```
[[0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0]]
```

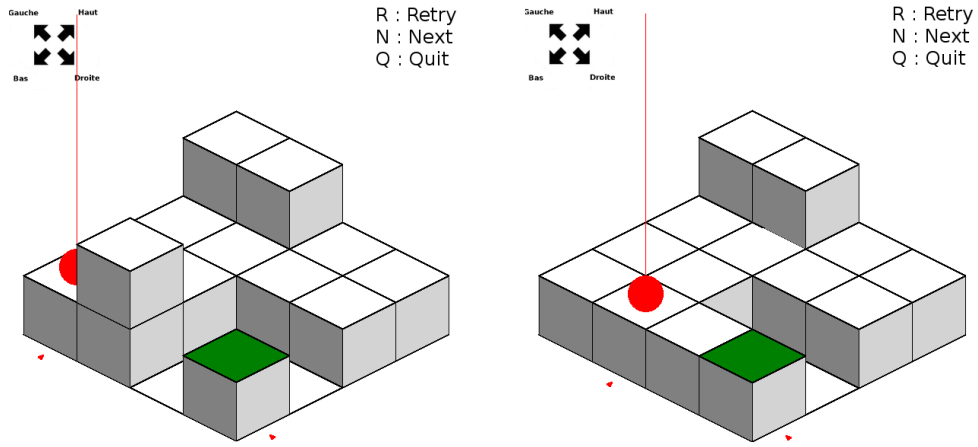
Une fois la sortie atteinte (case d'indice `[4][4]`), la partie n'a pas lieu de continuer.

4.2 Pousser un bloc

La bille peut dans certains cas pousser un unique bloc devant elle. Pour cela, le bloc à pousser doit être à la même hauteur que la bille et il doit y avoir de la place derrière le bloc poussé : la

hauteur de la case située derrière le bloc poussé doit être inférieure ou égale à la hauteur de la case où se trouve la bille. Si la hauteur est la même, le bloc est juste poussé en restant à la même hauteur. Si la hauteur est strictement plus petite, alors le bloc poussé "tombe". Dans ce cas la hauteur de la case où tombe le bloc augmente de 1. Il est bien sûr impossible de pousser un bloc en-dehors du plateau.

Dans l'exemple suivant, la boule s'apprête à pousser l'unique bloc déplaçable de la grille. La bille est d'abord partiellement cachée par le bloc qu'elle va pousser. Le joueur enfonce alors la touche flèche droite de son clavier. Le bloc se déplace vers la droite et tombe d'une hauteur de 1 formant ainsi un pont vers l'arrivée qui elle-même est aussi à hauteur 1.



4.3 Autres commandes de l'interface

Voici une liste des autres touches actives dans votre jeu :

- **La touche R** comme "Réessayer" doit remettre le niveau dans son état initial. Le joueur et les blocs déplacés doivent revenir à leur position de départ.
- **Les touches P et N** comme "Previous" et "Next" doivent permettre de passer au niveau précédent ou au niveau suivant.
- **La touche A** comme "Annuler" doit permettre d'annuler le dernier déplacement (déplacement simple ou poussée). Une poussée annulée doit remettre le bloc poussé à son ancienne place. Plusieurs appuis successifs sur cette touche annulent les derniers déplacements dans l'ordre inverse où ils ont été faits.
- **La touche Q** comme "Quitter" doit interrompre le jeu et fermer la fenêtre.

5 Génération aléatoire de grilles

Générer aléatoirement des grilles, et encore mieux des grilles intéressantes, n'est pas aussi simple qu'on pourrait le penser. En effet, si on lançait simplement un dé à 6 faces pour déterminer la hauteur de chaque case, on pourrait facilement fabriquer des grilles infaisables. Par exemple, les deux grilles suivantes sont infaisables :

```
((0, 4, 0, 0),        ((0, 0, 0, 0),
 (4, 4, 0, 0),         (0, 0, 0, 0),
 (0, 0, 0, 0),         (0, 0, 0, 0),
 (0, 0, 0, 0))         (0, 0, 0, 3))
```

Dans la première grille, la boule ne peut sortir de sa position de départ, dans la seconde grille, la bille peut aller partout mais ne montera jamais à la hauteur de la sortie.

Il faut donc contrôler l'aléatoire ! On peut s'autoriser des petites différences de hauteur mais pas trop. Il faut toujours tenter d'avoir un point de départ suffisamment haut pour aller jusque la sortie surtout si la grille est grande. Et bien sûr, il faut s'assurer que la grille fabriquée aléatoirement peut être résolue, cela, c'est le solveur qui va nous le dire (voir le paragraphe suivant).

Une bonne grille est souvent une grille difficile où le joueur doit réfléchir pour la résoudre. S'il existe un chemin clair en ligne droite du départ vers l'arrivée, la carte n'est pas très amusante. Par contre, si le joueur est obligé de pousser un ou plusieurs blocs pour atteindre la sortie alors la grille est intéressante. Aussi, quand les erreurs de mouvement forcent le joueur à recommencer (pas de droit à l'erreur), alors la grille est ambitieuse. Générer des grilles aléatoires difficiles peut rendre le jeu intéressant.

6 Recherche automatique de solutions

Comme l'algorithme du sac à dos tente de remplir un sac d'une taille donnée avec des objets donnés en prenant ou pas chaque objet, ici on peut implanter une fonction qui tente d'atteindre la case d'arrivée depuis le départ en tentant à chaque étape toutes les directions qui donnent un coup possible. Cette fonctionnalité est souvent désignée par le mot anglais *solver*. On pourrait parler en français d'une procédure automatique de résolution.

6.1 Algorithme de backtracking

Un algorithme naïf pour déterminer l'existence de solutions utilise le principe de *backtracking* vu en cours. En voici une description :

- Notons $C = (M\,, p)$ la configuration courante du jeu, constituée de la matrice des hauteurs actuelle M et de la position courante de la bille p.
- Notons V l'ensemble des configurations déjà visitées.
- Si $p = (n-1\,, n-1)$ est la position de la case d'arrivée, répondre "vrai" (il existe un chemin du départ à l'arrivée).
- Sinon, pour chaque direction $d \in \{bas\,, droite\,, haut\,, gauche\}$, si un déplacement dans la direction d est possible :
  1. Calculer la nouvelle configuration $C' = (M'\,, p')$ obtenue en se déplaçant dans la direction d ;
  2. Si la configuration C' appartient à V (a déjà été visitée), passer à la direction suivante ;
  3. Sinon, relancer la recherche à partir de C' ;
  4. Si aucune solution n'est trouvée, passer à la direction suivante.
- Si la recherche échoue dans toutes les directions, répondre "faux" (il n'existe pas de chemin).

6.2 Stocker une matrice dans un ensemble

Une difficulté de cet algorithme est de représenter l'ensemble V des configurations déjà visitées. On peut utiliser un ensemble Python (type `set`), mais un tel ensemble ne peut contenir que des valeurs *immutables*, l'interpréteur Python renvoie donc une erreur :

```
>>> plateau = [[3, 2, 1], [1, 2, 1], [1, 0, 1]]
>>> visites = set()
>>> visites.add(plateau)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Il faut donc préalablement convertir la matrice en tuple de tuples, par exemple à l'aide de la fonction (fournie) suivante :

```
def vers_tuple(liste_de_listes):
  return tuple(tuple(ligne) for ligne in liste_de_listes)
```

On peut ensuite utiliser le résultat de cette fonction pour ajouter la matrice à un ensemble :

```
>>> t_plateau = vers_tuple(plateau)
>>> visites = set()
>>> visites.add(t_plateau)
>>> print(visites)
{((3, 2, 1), (1, 2, 1), (1, 0, 1))}
```

Attention, dans votre solveur il ne suffira pas seulement d'enregistrer la matrice du plateau dans l'ensemble des configurations visitées, mais aussi associer la position de la bille !

6.3 Pour aller plus loin (optionnel)

On va rapidement se rendre compte que cet algorithme naïf est de complexité trop élevée et sera incapable de traiter les niveaux de taille importante. D'autre part, il ne fournit que des réponses de type "vrai ou faux", indiquant si oui ou non il existe une solution, mais il ne permet pas, par exemple, d'obtenir la suite des déplacements à effectuer ou de déterminer la solution la plus courte. Voici quelques possibilités d'amélioration (optionnelles) :

- Il est assez facile de déterminer la longueur de la solution trouvée. Il suffit pour cela de modifier légèrement l'algorithme afin de renvoyer un entier et non un booléen... mais cela ne fournira pas la solution la plus courte.
- Pour pouvoir renvoyer la solution trouvée à la fin d'un calcul, on peut enregistrer dans une liste la suite des déplacements essayés jusqu'ici. Avant un appel récursif, on ajoute une direction à cette liste. Si aucune solution n'est trouvée, on remplace cette dernière direction par la direction suivante.
- Si la bille se trouve sur une case dont l'altitude est strictement inférieure à celle de la case d'arrivée, on peut simplement abandonner la recherche à partir de cette position.

- Pour explorer les suites de déplacements par ordre de longueur croissante, on peut utiliser une *file* dans laquelle on stockera au fur et à mesure les configurations à explorer. Plutôt que de faire des appels récursifs, on sort la première configuration se trouvant dans la file, puis on ajoute ses voisins à la file, et ainsi de suite. Cette technique permet de déterminer une solution de longueur minimale à un niveau.

Une fois un solveur basique implémenté, on peut en faire plusieurs utilisations intéressantes, par exemple :

- Après avoir tiré un niveau aléatoire, on peut vérifier qu'il a bien au moins une solution. Si ce n'est pas le cas, on le rejette.
- On peut écrire deux versions du solveur : une version qui n'autorise pas les poussées de blocs, l'autre qui les autorise. De cette façon, si le premier solveur ne trouve pas de solution mais que le second en trouve une, c'est qu'il est obligatoire de pousser au moins un bloc pour réussir. Cela veut donc dire que la grille est peut-être intéressante...
- S'il est impossible de gagner depuis la position courante, on peut indiquer au joueur qu'il a perdu.

Avec un solveur plus perfectionné capable de calculer une solution de longueur minimale, on peut faire encore plus de choses :

- Si le solveur peut déterminer la solution la plus courte à un niveau, on peut utiliser cette information pour déterminer si le joueur a fait un score parfait.
- On peut rechercher automatiquement les niveaux les plus difficiles d'une taille n donnée, c'est à dire ceux dont la solution la plus courte est de longueur maximale.
- On peut ajouter une fonction "conseil" indiquant au joueur quel bloc il doit pousser ensuite.

Ces idées font réfléchir et sont amusantes, mais certaines d'entre elles sont assez difficiles à réaliser, même pour un programmeur expérimenté. Ne vous y aventurez qu'à vos risques et périls !

7 Amélioration possibles

En premier lieu, toute amélioration est envisageable à conditions d'en discuter avec un de vos enseignants. Il serait dommage que ce que vous considérez comme un plus soit considéré comme un moins par les correcteurs. Une prise de décision qui rend votre travail beaucoup plus facile a peu de chance d'être appréciée. Voici toutefois une liste de suggestions déjà validés par les enseignants :

- ★ Constituer un pack de niveaux "faits main".
- ★ Améliorer l'interface graphique avec des couleurs sympathiques.
- ★★ Rajouter deux compteurs pour indiquer le temps de jeu en secondes sur chaque grille, et le nombre de coups effectués.
- ★★ Ajouter la possibilité de pousser des empilements de blocs de taille supérieure à 1.

- ★★ Gérer des scores (sauvegardés dans un fichier). Le joueur donne son nom en début de programme, joue puis ses résultats sont intégrés aux scores en fin de programme. Un score est enregistré si le joueur a réussi à résoudre le niveau en moins de coups que le record précédemment établi.
- ★★ Ajouter une page d'accueil, un menu, des messages d'information (par exemple quand un niveau est gagné), des indications de progression (nom du niveau, nombre de niveaux restants).
- ★★ (long) Gérer la progression du jeu et le parcours du joueur. Les niveaux sont répartis en chapitres de difficulté croissante. Chaque niveau réussi se voit assorti d'une, deux ou trois étoiles en fonction de la longueur de la solution trouvée (1 étoile : solution médiocre, 3 étoiles : solution parfaite). Le nombre de coups donnant droit à chaque étoile peut être inscrit dans le fichier de sauvegarde, dans un format à définir. Le joueur déverrouille des niveaux en terminant les niveaux précédents.
- ★★ Structurer le code à l'aide de classes Python.
- ★★ Permettre de changer facilement la taille de la fenêtre du jeu ou d'autres paramètres à l'aide d'options en ligne de commande.
- ★★ Inclure un point de départ éventuellement différent du coin haut gauche dans la description de chaque niveau.
- ★★★ Changer la bille en un personnage dessiné qui s'oriente suivant son dernier mouvement.
- ★★★ Rajouter des animations. On voit la boule qui roule progressivement (ou le personnage qui marche) au lieu de voir des images figées de chaque état du jeu. Les blocs glissent et tombent de façon fluide.
- ★★★★ Modifier le jeu et le solveur afin qu'ils prennent en compte un coût différent pour chaque action (par exemple 1 pour un déplacement, 2 pour la poussée d'un bloc...).