

# Génération de labyrinthes tridimensionnels

Mathias ALOUI<sup>1</sup>, Jacques BADAGBON<sup>2</sup>, Roland BERTIN-JOHANNET<sup>3</sup>, Maxime POVEDA<sup>4</sup>, and Dyhia RAHNI<sup>5</sup>

- 1 Aix-Marseille Université, France, [mathias.aloui@etu.univ-amu.fr](mailto:mathias.aloui@etu.univ-amu.fr)
- 2 Aix-Marseille Université, France, [komlan.badagbon@etu.univ-amu.fr](mailto:komlan.badagbon@etu.univ-amu.fr)
- 3 Aix-Marseille Université, France, [roland.bertin-johannet@etu.univ-amu.fr](mailto:roland.bertin-johannet@etu.univ-amu.fr)
- 4 Aix-Marseille Université, France, [maxime.poveda@etu.univ-amu.fr](mailto:maxime.poveda@etu.univ-amu.fr)
- 5 Aix-Marseille Université, France, [dyhia.rahni@etu.univ-amu.fr](mailto:dyhia.rahni@etu.univ-amu.fr)

---

## Résumé

Nous nous posons le problème de générer des labyrinthes non pas bidimensionnels mais tridimensionnel. Nous étudions d'abord les algorithmes existant de génération en 2D et en listons les avantages et inconvénients. Puis nous discutons de l'aspect tridimensionnel, les formes et déplacements possibles, et définissons une architecture originale qui en plus motive notre choix d'algorithme. Ensuite, nous parlons des choix d'implémentation que nous avons faits. Puis, nous expliquons comment nous avons utilisé Blender afin d'afficher les fruits de notre travail. Enfin nous offrons des perspectives d'améliorations de celui-ci.

**Mots clés** Labyrinthe 3D ; Cube ; Blender ; Krustal ; Union-Find

## 1 Introduction

Les labyrinthes apparaissent dans toutes les cultures connues, quels que soient l'époque et le lieu. Cependant, la plupart du temps, on a affaire à des labyrinthes en deux dimensions. Passer de la deuxième à la troisième dimension dans un labyrinthe, c'est le problème que nous nous posons dans ce projet.

Tout d'abord, nous clarifions la notion commune de labyrinthe et introduisons celle qui nous intéresse : les labyrinthes parfaits. Ensuite, nous explorons trois différentes possibilités algorithmiques pour la génération d'un labyrinthe en deux dimensions : par les arbres couvrants de graphes, par exploration exhaustive ou par fusion aléatoire de chemins. Voulant représenter des labyrinthes intuitifs et équilibrés, nous faisons le choix de la fusion aléatoire de chemins, choix qui sera d'autant plus motivé plus tard.

Dans la suite, nous discutons la notion de labyrinthe tridimensionnel. Tout d'abord, nous l'explicitons. Puis, nous nous intéressons aux différentes formes possibles et aux agencements qu'elles offrent. À la lumière de ces explications, plusieurs questions apparaissent, auxquelles nous répondons : nous modéliserons un labyrinthe cubique en le considérant comme des labyrinthes 2D qui seront générés séparément puis « empilés » pour former un cube. Nous verrons que cette approche conforte encore plus le choix de notre algorithme.

Des considérations plus techniques sont ensuite faites en rapport à la structure utilisée. De plus, nous justifions notre choix d'utilisation du langage de programmation Python.

Une fois le labyrinthe 3D créé, il nous reste à trouver un moyen de l'afficher. C'est là la dernière partie de notre travail. Nous faisons le choix d'utiliser Blender, un logiciel pratique et complet. Ce logiciel répond à nos attentes en ce qu'il propose des outils de scripting avec la bibliothèque `blenderpy`, rendant la transition code/image plus aisée. Notre méthode d'utilisation de Blender pour afficher le labyrinthe est alors expliquée en détail.

Finalement, nous présentons nos résultats avant de proposer des pistes d'amélioration de notre travail.



## **2 Algorithmes de génération de labyrinthes bi-dimensionnels rectangulaires**

### **2.1 La notion de labyrinthes bi-dimensionnels**

Un labyrinthe, du latin *labyrinthus* qui signifie "enclos de bâtiments dont il est difficile de trouver l'issue", est une construction munie d'un point de départ, d'un point d'arrivée et d'un chemin plus ou moins complexe reliant les deux. Les labyrinthes ont eu plusieurs formes et utilités à travers l'histoire[9]. En mathématiques, les notions de génération et de résolution de labyrinthe ont déjà été étudiées. C'est ainsi que l'on distingue les labyrinthes parfaits et les labyrinthes imparfaits[10] :

- Un labyrinthe est dit parfait s'il existe un unique chemin entre n'importe quelles deux positions dans le labyrinthe. Si l'on représente ces positions par des sommets dans un graphe et les déplacements possibles par les arêtes, alors un labyrinthe parfait est un arbre.
- Un labyrinthe est dit imparfait dans le cas contraire. Ce type de labyrinthe peut contenir des cycles ou des îlots, c'est-à-dire plusieurs chemins entre deux positions et des positions inaccessibles.

Dans la suite, nous ne parlerons que des labyrinthes parfaits. Cependant, les aspects qui seront abordés, notamment la génération, peuvent être adaptés pour les labyrinthes imparfaits. Nous appellerons un labyrinthe sans aucun chemin un labyrinthe complet. C'est un labyrinthe où toutes les positions sont isolées, entourées de murs. Le graphe correspondant n'a aucune arête.

### **2.2 Les algorithmes classiques pour les labyrinthes**

Tous les algorithmes suivants génèrent un labyrinthe parfait à partir d'un labyrinthe complet.

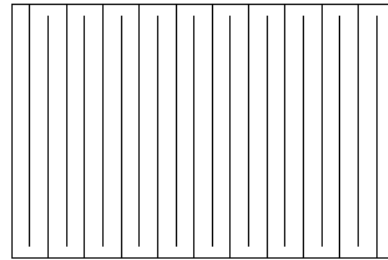
#### **2.2.1 Par les graphes**

Une première approche pour la génération de labyrinthes est d'utiliser les algorithmes de génération d'arbres couvrants pour les graphes. En effet, on peut représenter un labyrinthe sans murs internes, où tous les chemins sont possibles, sous forme d'une grille. Générer un chemin unique entre tous les points du labyrinthe revient alors à construire un arbre couvrant de cette grille. Les arêtes de l'arbre couvrant forment alors le chemin unique reliant toutes les positions dans le labyrinthe. Il existe pour cela de nombreux algorithmes comme le parcours en largeur, le parcours en profondeur ou encore l'algorithme de Prim[3].

Les deux premiers, sans modification, génèrent des labyrinthes peu intéressants. Les parcours en profondeur et en largeur vont tout le temps générer le même type de labyrinthe sous forme de "serpentin". Il est donc préférable d'utiliser un algorithme plus complexe tel que l'algorithme de Prim.



■ **Figure 1** Exemple parcours en largeur



■ **Figure 2** Exemple parcours en profondeur

L'algorithme de Prim est un algorithme glouton qui calcule un arbre couvrant minimal. Il consiste, à partir d'un sommet initial, à créer un arbre couvrant en ajoutant un à un les sommets accessibles en choisissant les arêtes de poids minimal.

Cet algorithme permet, avec une répartition aléatoire des poids sur les arêtes du graphe, de générer une plus grande variété de labyrinthes. Cependant, l'équilibre de ces labyrinthes ne paraît pas évident. Malgré la distribution aléatoire des poids, l'algorithme évolue à partir d'un sommet. Ainsi le choix des arêtes, malgré les poids aléatoires, est lié au choix du sommet initial. Voulant explorer d'autres solutions n'utilisant pas la représentation en graphe des labyrinthes, nous n'avons pas vérifié l'équilibre réel des labyrinthes générés par cet algorithme.

On peut tout de même dénoter que cette approche par graphe est intéressante car la notion de dimension n'entre pas en jeu. En effet, un algorithme qui génère un arbre couvrant d'un graphe n'est pas contraint par la forme de celui-ci. Ainsi, quelle que soit la forme de notre labyrinthe, tant que qu'elle peut être représentée par un graphe connexe, on peut trouver un arbre couvrant et donc créer un labyrinthe parfait.

### 2.2.2 Par exploration exhaustive

Cet algorithme peut être appliqué à un graphe mais également à des représentations plus graphiques des labyrinthes, par exemple en cellules. C'est le côté intuitif d'une telle représentation et de l'algorithme qui nous intéresse ici.

Cette autre approche consiste à modifier l'algorithme de parcours en profondeur afin de "se balader" dans le labyrinthe en abattant les murs au fur et à mesure[4]. Nous allons appeler cet algorithme "l'algorithme d'exploration exhaustive"[10]. Le principe est de créer un chemin en se déplaçant de pas en pas en abattant des murs et de revenir sur ses pas lorsque ouvrir un mur crée un cycle. On continue jusqu'à ce que toutes les cellules soient visitées. On peut stocker le chemin parcouru dans une pile et si les cellules ont été visitées ou non dans une variable interne à chaque cellule.

**Algorithm 1** Exploration exhaustive

---

```

1: pile = pile vide
2: cells = liste des cellules
3: for cell in cells do
4:   cell.visited = false
5: end for
6: cell = Random(cells)
7: cell.visited = True
8: pile.push(cell)
9: while pile is not empty do
10:   cell = pile.peek()
11:   neighbors = cell.unvisitedNeighbors()
12:   if neighbors is empty then
13:     pile.pop()
14:   else
15:     nextcell = Random(neighbors)
16:     DestroyWallBetween(cell,nextcell)
17:     nextcell.visited = True
18:     pile.push(nextcell)
19:   end if
20: end while

```

---

Cette méthode est simple à comprendre, simple à mettre en oeuvre et demande relativement peu de mémoire si la taille de la pile est bien gérée. Néanmoins, les labyrinthes générés ne sont là non plus pas équilibrés. Cet algorithme va créer des chemins jusqu'à arriver à une impasse, donc plus tôt le chemin est généré plus il a de chances d'être long. Dans notre cas, nous n'allons pas générer de très grands labyrinthes, la mémoire et la simplicité de l'algorithme ne sont donc pas de fortes contraintes.

### 2.2.3 Par fusion aléatoire de chemins

Cet algorithme comme le précédent peut être appliqué à d'autres représentations que celle par les graphes.

L'algorithme suivant est inspiré de l'algorithme de Kruskal[4][10]. L'algorithme de Kruskal est, comme l'algorithme de Prim cité précédemment, un algorithme d'arbre couvrant à poids minimal. Le principe est de sélectionner les arêtes de façon croissante selon leur poids et de les ajouter si elles ne créent pas de cycle.

Du point de vue du labyrinthe, cela revient à enlever aléatoirement des murs jusqu'à ce que toutes les cellules soient reliées par un chemin. Pour ce faire, toutes les cellules ont un représentant unique (elles-mêmes initialement). On choisit aléatoirement un mur, et si les deux cellules de part et d'autre ont un représentant différent, alors on enlève le mur et on met le même représentant pour les deux cellules. Nous appelons cet algorithme "fusion aléatoire de chemins" :

---

**Algorithm 2** Fusion de chemins

---

```

1: murs = liste des murs
2: cells = liste des cellules
3: for cell in cells do
4:   cell.représentant = cell
5: end for
6: while  $\exists c1, c2 \in cells | Find(c1) \neq Find(c2)$  do
7:   m = murs.popRandom()
8:   c1 = FirstSide(m)
9:   c2 = OtherSide(m)
10:  if Find(c1)  $\neq$  Find(c2) then
11:    Union(c1, c2)
12:  end if
13: end while

```

---

Pour implémenter la suppression d'un mur, la structure de données Union-Find est tout indiquée. Elle consiste en deux opérations : Union, réunir deux ensembles, et Find, trouver à quel ensemble appartient l'élément. Des améliorations existent pour permettre de réduire le temps d'exécution d'Union et de Find permettant d'améliorer les performances globales de notre algorithme.

La fusion aléatoire de chemins permet d'avoir des labyrinthes équilibrés. En effet, tous les murs ont la même probabilité d'être choisis à un instant donné, cela implique également la génération de plus de bifurcations que par l'algorithme d'exploration exhaustive. Cela donne également des possibilités de modifier ces probabilités pour générer des labyrinthes particuliers. Cette particularité, bien que peu exploitée par la suite, est un avantage certain.

Du côté vitesse d'exécution et mémoire, l'utilisation de la structure de données Union-Find et la taille relativement faible des labyrinthes que nous allons générer permet de bonnes performances, similaires à l'algorithme d'exploration exhaustive. Seules les considérations sur l'équilibre du labyrinthe et des améliorations possibles sont donc nécessaires.

Pour l'instant, l'algorithme de fusion aléatoire de chemins semble être le plus adapté. Il nous permet une plus grande liberté sur la représentation du labyrinthe qui reste encore à définir, surtout pour l'aspect tridimensionnel, offre une génération plus équilibrée que celle de l'exploration exhaustive et peut être facilement adapté pour modifier l'allure des labyrinthes générés.

### 3 Concept de labyrinthe tridimensionnel

De façon générale, un labyrinthe 2D, quelle que soit sa forme (carré, triangle, disque...), peut être vu comme une structure dans le plan bidimensionnel où il n'existe pas ou peu de chemins directs et courts entre deux points internes  $A$  et  $B$  donnés. Ainsi "résoudre" le labyrinthe revient à trouver une suite de déplacements unitaires réalisables partant d'un point initial  $I$  donné, jusqu'à un autre point  $S$  (sortie), suivant les directions vectorielles du repère 2D considéré. La difficulté augmente selon l'éloignement des points considérés et la complexité du labyrinthe (taille du labyrinthe, présence ou non de cycles et de sous-espaces cloisonnés...). Les labyrinthes générés dans ce projet étant parfaits, ils assurent une meilleure fluidité des déplacements lors de la recherche de la sortie  $S$ .

Un labyrinthe 3D respecte pratiquement les mêmes règles qu'un labyrinthe 2D. On y ajoute seulement une nouvelle direction passant d'un labyrinthe dans un plan à un labyrinthe dans l'espace tridimensionnel, offrant ainsi la possibilité de faire de nouvelles formes de labyrinthes (cube, parallélépipède, pyramide, prisme, sphère...) et des déplacements vectoriels suivant trois axes.

Il est donc primordial, avant l'implémentation de l'algorithme de génération, de définir un modèle conceptuel pour le labyrinthe. Il va influencer le choix de l'algorithme, des structures de données utilisées et de leurs implémentations. Pour un choix optimal du modèle, il est important de tenir compte des caractéristiques visuo-spatiales que doit présenter le labyrinthe 3D.

#### 3.1 Formes et déplacements

Comme dit plus tôt, selon la forme 3D considérée, les propriétés et déplacements dans le labyrinthe varient. Ainsi, avant de se lancer dans la génération d'une forme donnée, on peut considérer les perspectives suivantes :

- Pour un labyrinthe cubique : intuitivement, les déplacements sont possibles selon deux directions dans le plan (avant/arrière et gauche/droite) ; ils correspondent aux médianes de la face que l'on observe auxquelles s'ajoute la possibilité de se déplacer dans une troisième direction, celle de la normale au plan. Cela vaut également pour les labyrinthes de formes parallélépipédiques et cylindriques non circulaires[7].
- Pour un labyrinthe sphérique : les déplacements peuvent se faire suivant des arcs de cercle concentriques à la sphère mais également de façon centripète. Le même principe s'applique aux cylindres circulaires avec des déplacements selon des arcs circulaires ayant leurs centres sur l'axe de révolution et suivant la direction de ce dernier[8].
- Pour un labyrinthe pyramidal : les directions peuvent être les médianes de chaque face ainsi que celles des vecteurs normaux à celles-ci. Ce même principe s'applique aux prismes et autres polyèdres[11].

À présent que nous avons pris connaissance de configurations possibles pour certaines formes tridimensionnelles, on peut s'intéresser à un premier modèle pour notre projet. La première figure que nous allons explorer, et qui sera détaillée dans ce rapport, est le cube. Cette figure étant régulière, ses déplacements internes se font suivant des directions orthogonales et ce indépendamment du choix de la face. Néanmoins, il existe divers moyens de le modéliser. Il reste maintenant à affiner notre choix en fonction des caractéristiques du cube précédemment énoncées et de la contrainte de pouvoir observer en temps réel de l'extérieur les déplacements effectués dans le labyrinthe pour le résoudre.

### 3.2 Architecture en demi-étages

Diverses possibilités sont à considérer dans la conception d'un labyrinthe cubique. On peut par exemple envisager :

- Un labyrinthe en un bloc : c'est-à-dire que toutes les directions sont possibles, le labyrinthe est "sans contrainte". L'inconvénient avec ce modèle est que même en jouant sur l'opacité des murs, il sera difficile d'observer les déplacements à l'intérieur du labyrinthe.
- Un labyrinthe où seul l'extérieur du cube est accessible n'exploite pas assez l'aspect tridimensionnel et limite grandement les possibilités du labyrinthe.

Pour pallier à tout cela, un modèle plus flexible et évolutif a été développé pour ce projet : une architecture cubique à étages. Le principe est de voir le cube comme une succession de pavés droits à base carrée et de hauteur unitaire. "L'empilement" des pavés droits forme le cube. Chaque pavé droit peut être vu comme un labyrinthe indépendant soit 2D soit 3D. Nous avons choisi de faire des étages en 3D.

Chaque étage est en réalité la superposition de deux demi-étages. Chaque demi-étage est un labyrinthe 2D. Nous avons un premier trou, l'entrée de l'étage, au milieu de la face supérieure de l'étage (la face supérieure du premier demi-étage). Puis nous avons un autre trou, le point de passage, au niveau de l'un des bords de la face inférieure du premier demi-étage (respectivement la face supérieure du deuxième demi-étage). Enfin nous avons un dernier trou, la sortie de l'étage, au milieu de la face inférieure de l'étage (la face inférieure du deuxième demi-étage). La résolution d'un étage consiste donc à trouver un chemin entre l'entrée et le point de passage dans le premier demi-étage puis un chemin du point de passage à la sortie dans le deuxième demi-étage. Ainsi, quand on passe d'un étage à l'autre (en supposant que l'on évolue du haut vers le bas dans le cube assemblé), on accède d'abord au demi-étage du haut, puis, en le résolvant, on accède au demi-étage du bas. Résoudre le labyrinthe revient donc à résoudre tous ses étages ou plus précisément tous ses demi-étages.

Les avantages de ce modèle sont : sa simplicité de résolution, étage par étage, la facilité de se repérer, car les déplacements sont limités dans un plan à chaque étage donné, la possibilité d'enlever les étages déjà résolus, afin d'avoir une meilleure visibilité, et sa modularité. La modularité est possible car les entrées et sorties des étages sont toujours au centre. Il est donc possible d'intervertir n'importe quels étages entre eux. Un cube n'est donc pas un seul labyrinthe mais plusieurs labyrinthes pouvant être créés en modifiant l'ordre des étages.

### 3.3 Choix de l'algorithme de génération

L'architecture proposée se base donc sur une génération de labyrinthes 2D standards. Tous les algorithmes cités en première partie sont donc applicables. On va préférer les algorithmes ne se basant pas sur la représentation en graphe afin de rester libre dans nos choix. De plus, on a déjà vu que, dans notre cas, l'algorithme de fusion aléatoire de chemins est meilleur que celui d'exploration exhaustive. Notre choix se confirme donc sur l'utilisation de la fusion aléatoire de chemins.

## 4 Méthodes d'implémentation

Pour implémenter notre architecture, il faut d'abord choisir un langage de programmation adapté. En l'occurrence, Python s'avère être intéressant grâce à ses bibliothèques étendues et variées, notamment Numpy[2]. Il est également simple à comprendre et à utiliser, cependant, son utilisation est assez restreinte dans le domaine du design et de la représentation graphique, c'est pour cela que nous allons l'utiliser uniquement pour représenter nos données.

Après avoir envisagé une architecture et un langage de programmation, il est maintenant nécessaire d'expliquer la méthode utilisée pour la représentation et l'implémentation des composantes du labyrinthe.

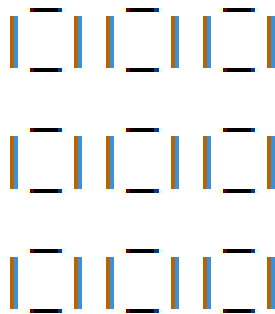
Il existe plusieurs manières de représenter les demi-étages du labyrinthe 3D :

- Représenter chaque demi-étage comme une succession de murs : les murs seront agencés de façon à ce que l'ensemble forme un labyrinthe 2D, le demi-étage. Les trous qui représentent les points de passage d'un demi-étage à l'autre seront, dans ce cas-là, disposés dans leurs emplacements respectifs à la place d'un mur. Cette représentation n'est cependant pas pratique dans le sens où les composantes du demi-étage sont considérées individuellement et l'implémentation ne se fera pas de façon naturelle.
- Représenter chaque demi-étage comme une succession de cellules : cette méthode est particulièrement intéressante, chaque cellule étant un objet composé de murs et pouvant comporter un trou. Pour résoudre le demi-étage, il faut trouver un chemin en se déplaçant de cellule en cellule jusqu'à trouver le trou. Contrairement à la première méthode, le demi-étage n'est plus considéré comme une collection de murs mais comme un ensemble de cellules qui ont des murs. Ainsi, l'implémentation de l'algorithme de fusion aléatoire de chemins en sera facilitée.

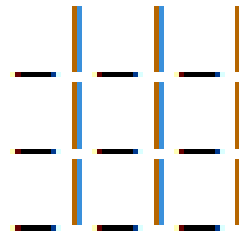
En considérant la méthode des cellules, il faut, pour pouvoir fixer les composantes de ces dernières, prendre en compte le facteur de redondance des murs. En effet, si chaque cellule comporte ses quatre murs (pour chaque direction), alors chaque mur interne est représenté dans les deux cellules qu'il sépare : il est représenté en double. Il faut aussi pouvoir définir quels murs sont destructibles et, le cas échéant, pouvoir les rendre destructibles si besoin. Ces problèmes sont capitaux lors de l'implémentation.

La solution adoptée est de ne représenter que deux murs contigus dans chaque cellule au lieu de quatre (suivant les 4 directions). On choisit arbitrairement les murs de droite et du bas. En générant moins de composantes, on évite le problème de redondance des murs ce qui permet d'économiser du temps et de la mémoire. De plus, chaque cellule peut contenir une variable sur sa nature notamment sur la destructibilité de ses murs. Cette approche résout donc les problèmes posés.





■ **Figure 3** Labyrinthe 2D de taille 3 produit avec des cellules contenant 4 murs



■ **Figure 4** Labyrinthe 2D de taille 3 produit avec des cellules contenant 2 murs

On remarque que dans la figure 4 les bordures gauches et supérieures ne sont pas produites, cependant, cela ne pose aucun problème car ces bordures sont indestructibles et donc n'empêchent en aucun cas l'application de l'algorithme générateur du labyrinthe.

Après l'implémentation des cellules, il reste à implémenter l'étage. Comme expliqué dans les points précédents, un étage du labyrinthe tridimensionnel est composé de deux demi-étages, représentant chacun un labyrinthe 2D composé de cellules. Il faut considérer les deux demi-étages. Chaque demi-étage doit pouvoir être manipulé et modifié. Voici quelques solutions possibles :

- Représentation par graphe d'états : On peut voir chaque cellule comme étant un sommet dans un graphe. Cette méthode permet, pour la génération du labyrinthe, d'utiliser des algorithmes d'arbres couvrants[3]. Cette approche ne permet pas l'utilisation des cellules comme décrites précédemment avec seulement deux murs.
- Représentation par matrices de cellules : cette méthode permet de représenter notre demi-étage sous forme d'une matrice. On identifie chaque cellule par ses coordonnées dans cette matrice. Cependant, il nous est impossible de travailler directement sur des matrices en utilisant le langage Python, la bibliothèque Numpy ne permettant pas de créer des matrices d'objets. Il est cependant possible de travailler avec des listes imbriquées[5] qui utilisent les caractéristiques d'une matrice et qui permettent d'avoir les valeurs voulues, en l'occurrence des cellules.

On représentera donc chaque demi-étage par des listes imbriquées contenant nos cellules. Ces dernières pourront être récupérées grâce à leurs identifiants ainsi que leurs coordonnées. Cette méthode, contrairement à celle des graphes d'états, nous permet d'appliquer l'algorithme choisi.

En effet, on peut implémenter la structure de données Union-Find dans les cellules. On stocke dans chaque cellule son représentant, celui-ci étant une autre cellule. De plus, lors de la sélection aléatoire d'un mur, on peut d'abord choisir aléatoirement une cellule puis l'un de ses murs. Pour assurer que chaque mur ait la même probabilité d'être choisi, les cellules contenant deux murs destructibles (ne sont pas des bordures du labyrinthe) sont mises deux fois dans la liste. Celle-ci est ensuite mélangée et lorsqu'une cellule est choisie, elle est enlevée de la liste. L'algorithme se poursuit jusqu'à ce que la liste soit vide. De cette façon, tous les murs ne sont sélectionnés qu'une seule et unique fois.

## 5 Modélisation 3D et rendu

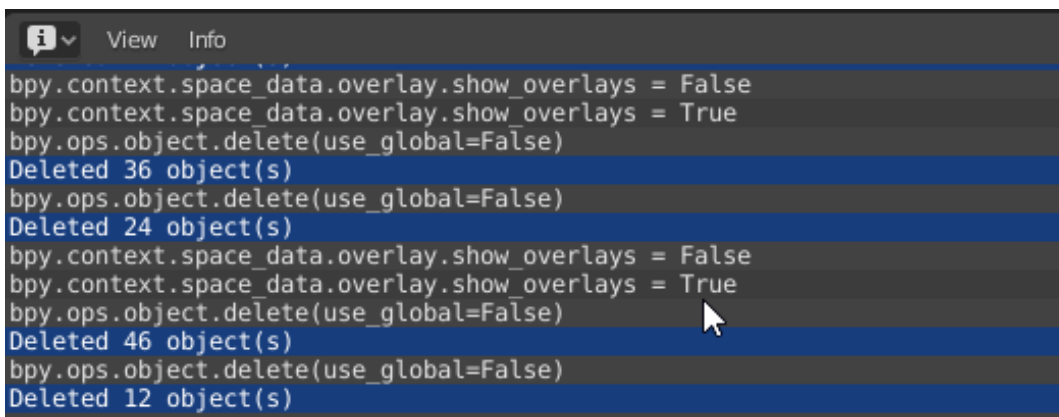
Nous avons maintenant la possibilité de générer des labyrinthes cubiques. La prochaine étape est de les visualiser sous la forme d'un objet manipulable. Pour cela, nous décidons d'utiliser un logiciel de création 3D car le temps prévu pour ce projet ne nous permet pas de créer notre propre interface d'éditeurs 3D. La première difficulté est de trouver un tel logiciel utilisant le langage Python et nous permettant donc d'utiliser notre code de génération de labyrinthes.

### 5.1 Utilisation du logiciel libre Blender

Nous nous sommes rapidement intéressé à Blender[1]. Il s'agit d'un logiciel très complet et très utilisé par les artistes d'animation, gratuit et cross-platform qui marche sous Windows, Linux et Macintosh. Il supporte l'intégralité du pipeline 3D, c'est-à-dire la modélisation, l'animation, la simulation, le rendu, la composition, le suivi de mouvement et la création de jeux vidéo. Pour les plus avancés, il y a la possibilité d'utiliser un outil de scripting en python. C'est ce côté-là qui nous a permis de nous décider quant à l'emploi de ce logiciel.

Dans cet espace de scripting, nous utilisons la bibliothèque blenderpy (bpy)[6]. Celle-ci nous permet d'utiliser des fonctionnalités qui nécessitent normalement d'être faites "à la main", c'est-à-dire avec la souris ou le clavier, comme la génération ou la modification d'objets 3D dans l'espace d'édition. A la place, blenderpy nous offre des fonctions qui automatisent ces procédés.

Blender possède aussi un espace "info" qui indique dans une pseudo-console toutes les actions portées dans le logiciel. Il suffit de copier la ligne qui correspond à l'action qui nous intéresse et de la coller dans le script pour traduire une action en code, sous réserve de certaines conditions. Par exemple, certaines actions nécessitent qu'un objet soit sélectionné, si ce n'est pas le cas, la ligne de code correspondante n'aura aucun effet. Cet espace est l'un des outils principaux que nous utilisons afin d'automatiser la construction de notre objet. C'est ici que nous avons récupéré le code que nous avons utilisé par la suite.



```
bpy.context.space_data.overlay.show_overlays = False
bpy.context.space_data.overlay.show_overlays = True
bpy.ops.object.delete(use_global=False)
Deleted 36 object(s)
bpy.ops.object.delete(use_global=False)
Deleted 24 object(s)
bpy.context.space_data.overlay.show_overlays = False
bpy.context.space_data.overlay.show_overlays = True
bpy.ops.object.delete(use_global=False)
Deleted 46 object(s)
bpy.ops.object.delete(use_global=False)
Deleted 12 object(s)
```

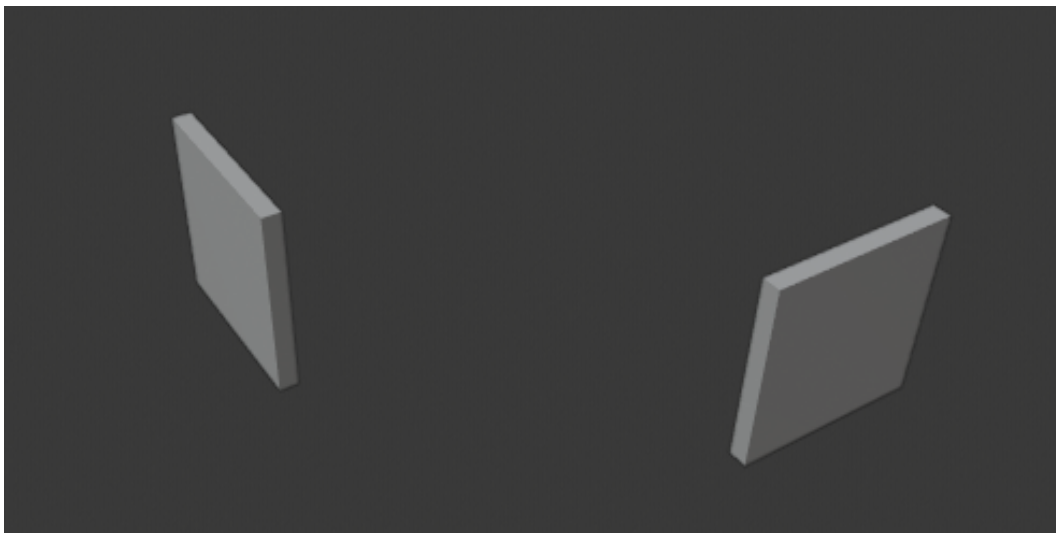
■ **Figure 5** Exemple d'actions affichés dans la pseudo-console

On peut voir ci-dessus la ligne `bpy.ops.object.delete(use_global=False)` qui correspond à la suppression des divers objets sélectionnés au moment de l'action.

## 5.2 Construction du labyrinthe dans Blender

La première piste consistant à générer un labyrinthe complet (voir 2.1) puis à creuser le chemin généré par notre algorithme est rapidement écartée. En effet, le fait d'extruder les murs là où ils ont été enlevés par l'algorithme et au niveau des entrées, sorties et points de passage des étages est complexe. Blender possède une fonctionnalité qui permet de détruire des morceaux d'objets. Mais, la complexité de celle-ci, liée à l'utilisation dans le script, rend cette dernière beaucoup trop chronophage à implémenter et compliquée nous forçant à repenser notre approche.

On commence d'abord par construire deux objets, un mur horizontal et un mur vertical grâce à deux fonctions du module bpy : une qui permet de générer un cube à des coordonnées données et une autre qui permet de modifier les dimensions d'un objet. Les murs obtenus sont des pavés droits. Ils sont identiques mais n'ont pas la même orientation dans l'espace.



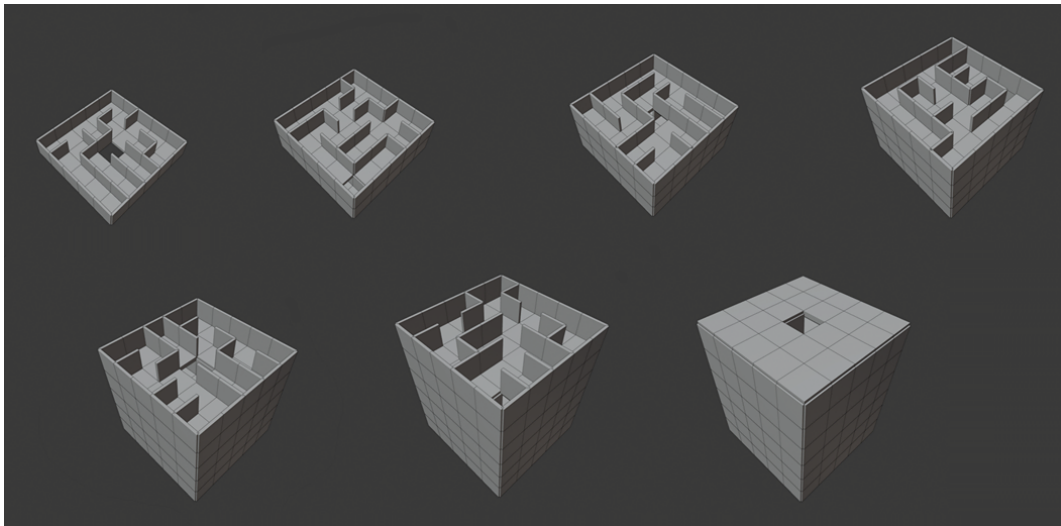
■ **Figure 6** Un mur vertical et un mur horizontal

Puis, pour chaque demi-étage, on utilise une fonction de conversion permettant à partir de notre architecture en cellules de récupérer deux matrices. L'une correspond aux murs verticaux du demi-étage, l'autre aux murs horizontaux. On peut ainsi savoir à tout instant la présence ou non d'un mur dans le demi-étage. Nous utilisons ces informations afin de placer correctement nos objets. La hauteur de génération dépend du demi-étage considéré. Il s'agit maintenant de créer les faces supérieures et inférieures des demi-étages en laissant un "trou" pour les entrées, sorties et points de passage. Si l'on voit chaque face inférieure des demi-étages comme un quadrillage, on crée un nouvel objet sol que l'on génère partout sauf à l'endroit du point de passage pour les demi-étages supérieurs et à l'endroit de la sortie pour les demi-étages inférieurs. Pour l'étage le plus haut, il ne reste plus qu'à générer sa face supérieure en laissant le trou de son entrée. La succession de ces procédés nous permet d'avoir un script qui, après exécution, nous donne un cube avec un point de départ, le trou au milieu de la face du dessus, et un point d'arrivée, le trou au milieu de la face du dessous.

Il est possible grâce au script Python d'importer le code permettant l'initialisation d'un labyrinthe et le code de notre algorithme. Le script, lorsqu'il est lu par Blender, nous permet alors de créer un cube, puis de générer le labyrinthe à l'intérieur et enfin de l'afficher en 3D en une seule exécution.

## 6 Résultats et améliorations possibles

Pour conclure, nous avons établi un concept de labyrinthe cubique ainsi qu'un algorithme pour le générer. Nous l'avons codé avec le langage Python. Puis, grâce au script Blender, nous avons trouvé le moyen d'afficher le labyrinthe. Voici un exemple de labyrinthe à six demi-étages (soit trois étages), affichés un par un. On peut supprimer les différents murs et sols afin de vérifier la consistance de notre cube.



■ **Figure 7** Exemple d'un labyrinthe généré par notre méthode, puis affiché demi-étage par demi-étage

On peut constater que chaque demi-étage est un labyrinthe parfait, qu'il n'y a pas de chemin extrêmement simple entre deux "trous" et qu'il y a une certaine diversité entre les étages. On peut s'imaginer à l'intérieur du labyrinthe, ne connaissant pas à l'avance l'emplacement des trous : on a affaire à des culs-de-sac, le chemin à suivre est imprévisible ; le labyrinthe remplit bel et bien son rôle.

Voyons maintenant des améliorations possibles à notre projet :

Une première amélioration simple est de trouver le bon niveau d'opacité des murs afin d'avoir une bonne visibilité sur le produit fini.

Nous pouvons aussi remarquer que malgré le côté modulaire de la construction du labyrinthe par "empilement" d'étages qui permet d'en faire plusieurs combinaisons, cette même méthode est limitante en matière de diversité : on se voit résoudre le labyrinthe étage par étage, là où on aurait pu imaginer faire des allers-retours entre les étages et même entre demi-étages pour plus de diversité.

De même, hormis la position du point de passage, nous n'avons pas de système permettant de vérifier la complexité de chaque demi-étage. De plus, nous n'avons généré que des labyrinthes parfaits, il est possible de modifier l'algorithme afin de créer encore plus de labyrinthes différents et notamment imparfaits. On peut également imaginer d'autres formes de labyrinthes utilisant la même architecture en étages.

Dans un objectif de commercialisation du produit, la modularité de l'architecture permet la création d'étages spéciaux ou à collectionner pour ensuite faire des cubes uniques et personnalisés.

---

**Références**

---

- 1 Blender. Home of the blender project - free and open 3d creation source, 2020. [En ligne; Page disponible le 20-avril-2020]. URL : <https://www.blender.org>.
- 2 David Cassagne. Introduction à numpy, 2019. [En ligne; Page disponible le 20-avril-2020]. URL : <https://courspython.com/apprendre-numpy.html>.
- 3 Laboratoire de Mathématiques, Université de Savoie. Génération et résolution de labyrinthes, 2017. [En ligne; Page disponible le 20-avril-2020]. URL : [https://www.lama.univ-savoie.fr/mediawiki/index.php/G%C3%A9n%C3%A9ration\\_et\\_r%C3%A9solution\\_de\\_labyrinthes](https://www.lama.univ-savoie.fr/mediawiki/index.php/G%C3%A9n%C3%A9ration_et_r%C3%A9solution_de_labyrinthes).
- 4 Laboratoire de Mathématiques, Université de Savoie. Génération et résolution de labyrinthes ii, 2019. [En ligne; Page disponible le 20-avril-2020]. URL : [https://www.lama.univ-savoie.fr/mediawiki/index.php/G%C3%A9n%C3%A9ration\\_et\\_r%C3%A9solution\\_de\\_labyrinthes\\_II](https://www.lama.univ-savoie.fr/mediawiki/index.php/G%C3%A9n%C3%A9ration_et_r%C3%A9solution_de_labyrinthes_II).
- 5 ESSADDOUKI Mostafa. Les matrices en python, 2019. [En ligne; Page disponible le 20-avril-2020]. URL : <https://developpement-informatique.com/article/229/les-matrices-en-python>.
- 6 TylerGubala. bpy - pypi, 2020. URL : <https://pypi.org/project/bpy/>.
- 7 Wikipédia. Cylindre — wikipédia, l'encyclopédie libre, 2020. [En ligne; Page disponible le 8-mai-2020]. URL : <http://fr.wikipedia.org/w/index.php?title=Cylindre&oldid=170601586>.
- 8 Wikipédia. Cylindre de révolution — wikipédia, l'encyclopédie libre, 2020. [En ligne; Page disponible le 7-mai-2020]. URL : [http://fr.wikipedia.org/w/index.php?title=Cylindre\\_de\\_r%C3%A9volution&oldid=170553104](http://fr.wikipedia.org/w/index.php?title=Cylindre_de_r%C3%A9volution&oldid=170553104).
- 9 Wikipédia. Labyrinthe — wikipédia, l'encyclopédie libre, 2020. [En ligne; Page disponible le 20-avril-2020]. URL : <http://fr.wikipedia.org/w/index.php?title=Labyrinthe&oldid=169813782>.
- 10 Wikipédia. Modélisation mathématique de labyrinthe — wikipédia, l'encyclopédie libre, 2020. [En ligne; Page disponible le 2-janvier-2020]. URL : [http://fr.wikipedia.org/w/index.php?title=Mod%C3%A9lisation\\_math%C3%A9matique\\_de\\_labyrinthe&oldid=165955332](http://fr.wikipedia.org/w/index.php?title=Mod%C3%A9lisation_math%C3%A9matique_de_labyrinthe&oldid=165955332).
- 11 Wikipédia. Polyedre — wikipédia, l'encyclopédie libre, 2020. [En ligne; Page disponible le 26-mars-2020]. URL : <http://fr.wikipedia.org/w/index.php?title=Poly%C3%A8dre&oldid=168825954>.

## **A** Explication de code

Vous pouvez retrouver le code source de notre projet à l'adresse :  
<https://github.com/L2InfoAMU/projetl3mi-production-de-labyrinthe-3d>.

### **A.1 Dir**

La classe `dir` permet de définir les directions dans le labyrinthe 2D.

#### **A.1.1 Variables**

1. `right = 0`
2. `down = 1`
3. `left = 2`
4. `top = 3`

#### **A.1.2 Fonctions**

1. `dir opposite()` : prend en argument une direction, retourne la direction opposée.

### **A.2 Cell**

La classe `Cell` s'occupe de définir et d'appliquer les propriétés des cellules de notre labyrinthe.

#### **A.2.1 Variables**

1. `walls = [boolean, boolean]` : représente deux murs de la cellule, si le mur existe le booléen est vrai sinon il est faux.
2. `destroyable = [boolean, boolean]` : représente si les murs sont destructibles, si oui le booléen est vrai sinon il est faux.
3. `partition = int` : représente dans quelle partition du demi-étage la cellule est située (prévu initialement pour créer des niveaux difficiles)
4. `id = int` : représente l'identifiant de la cellule.
5. `parent = Cell` : représente le représentant de la cellule (utilisé lors de la génération du labyrinthe)
6. `rank = int` : représente le poids dans la structure de données Union-Find (utilisé lors de la génération du labyrinthe)

#### **A.2.2 Fonctions**

1. `int getPartition()` : ne prend pas d'argument, retourne la variable interne `partition`.
2. `int getId()` : ne prend pas d'argument, retourne la variable interne `id`.
3. `void setId(int id)` : prend en argument un entier `id`, modifie la variable interne `id` par celle donnée en argument, ne retourne rien.
4. `Cell getParent()` : ne prend pas d'argument, retourne la variable interne `parent`.
5. `void setParent(Cell parent)` : prend en argument un `Cell parent`, modifie la variable interne `parent` par celle donnée en argument, ne retourne rien.

6. `int getRank()` : ne prend pas d'argument, retourne la variable interne `rank`.
7. `void setRank(int rank)` : prend en argument un entier `rank`, modifie la variable interne `rank` par celle donnée en argument, ne retourne rien.
8. `void increaseRank()` : ne prend pas d'argument, incrémente la variable interne `rank`, ne retourne rien.
9. `boolean hasWall(int wall)` : prend en argument un entier `wall`, teste si le mur correspondant à `wall` existe, retourne vrai si c'est le cas et faux sinon.
10. `void buildWall(int wall)` : prend en argument un entier `wall`, met à vrai le mur correspondant à `wall` dans la variable interne `walls`, ne retourne rien.
11. `boolean makeIndestructible(int wall)` : prend en argument un entier `wall`, teste si le mur correspondant à `wall` existe, si oui alors met à faux le mur correspondant à `wall` dans la variable interne `destroyable`, retourne vrai si c'est la cas et faux sinon.
12. `boolean makeRightBorder()` : ne prend pas d'argument, appelle la fonction interne `makeIndestructible` avec comme argument le mur de droite `Dir.right`, retourne la valeur retournée par `makeIndestructible`.
13. `makeBottomBorder()` : ne prend pas d'argument, appelle la fonction interne `makeIndestructible` avec comme argument le mur du bas `Dir.down`, retourne la valeur retournée par `makeIndestructible`.
14. `boolean isDestoyable(int wall)` : prend en argument un entier `wall`, teste si le mur correspondant à `wall` est destructible, retourne vrai si c'est le cas et faux sinon.
15. `boolean destroyWall(int wall)` : prend en argument un entier `wall`, teste si le mur correspondant à `wall` est destructible, si oui alors met à faux le mur correspondant à `wall` dans la variable interne `walls`, retourne vrai si c'est le cas et faux sinon.
16. `void print(boolean isHole = False)` : prend en argument optionnel un booléen `isHole`, teste si la variable `isHole` est à vrai, si oui alors affiche la cellule sur la sortie standard avec un trou sinon sans trou.

## A.3 Floor

### A.3.1 Variables

1. `top = Cell[ ][ ]` : (ligne/colonne), représente le premier demi-étage.
2. `bottom = Cell[ ][ ]` : (ligne/colonne), représente le deuxième demi-étage.
3. `size = int` : représente la taille de l'étage (nombre de cellules par ligne/colonne)
4. `holes = int[]` : représnte la liste des points de passage.
5. `next = Floor` : représente l'étage suivant (étage du dessous dans le cube)
6. `generated = boolean` : représente si l'étage a été généré par l'algorithme ou non.

### A.3.2 Fonctions

1. `Cell[ ][ ] getTop()` : ne prend pas d'argument, retourne la variable interne `top`.
2. `Cell[ ][ ] getBottom()` : ne prend pas d'argument, retourne la variable interne `bottom`.
3. `int getSize()` : ne prend pas d'argument, retourne la variable interne `size`.
4. `int[ ] getHoles()` : ne prend pas d'argument, retourne la variable interne `holes`.

5. boolean isGenerated() : ne prend pas d'argument, retourne la variable interne generated.
6. void makeGenerated() : ne prend pas d'argument, met la variable interne generated à vrai, ne retourne rien.
7. Floor getNextFloor() : ne prend pas d'argument, retourne la variable interne next.
8. void setNextFloor(Floor floor) : prend en argument un Floor floor, modifie la variable interne next par la variable floor, ne retourne rien.
9. (int,int)[] getCoordHoles() : ne prend pas d'argument, retourne une liste de tuples correspondant à la variable interne holes convertie en coordonnées, ne retourne rien.
10. void makeBorders() : ne prend pas d'argument, parcourt les cellules des variables internes top et bottom et crée les bordures inférieures et de droite du labyrinthe en utilisant les fonctions makeBottomBorder et makeRightBorder dans Cell, ne retourne rien.
11. void addHole(int hole) : prend en argument un entier hole, ajoute l'entier hole à la variable interne holes, ne retourne rien.
12. void setIdCells() : ne prend pas d'argument, parcourt les cellules des variables internes top et bottom et initialise les identifiants des cellules en utilisant la fonction setId dans Cell, ne retourne rien.
13. int computeId(int line, int column) : prend en argument deux entiers line et column, retourne l'identifiant de la cellule se trouvant aux coordonnées correspondant à la ligne line et à la colonne column.
14. (int,int) computeCoord(int id) : prend en argument un entier id, retourne les coordonnées de la cellule correspondant à l'identifiant id.
15. Cell getCell(int cellId, Cell[][] halfFloor) : prend en argument un entier cellId et un demi-étage halfFloor, retourne la cellule correspondant à l'identifiant cellId dans le demi-étage halfFloor.
16. boolean cellIsLeftBorder(int cellId) : prend en argument un entier cellId, teste si la cellule correspondant à l'identifiant cellId est sur une bordure de gauche, retourne vrai si c'est le cas et faux sinon.
17. boolean cellIsTopBorder(int cellId) : prend en argument un entier cellId, teste si la cellule correspondant à l'identifiant cellId est sur une bordure supérieure, retourne vrai si c'est le cas et faux sinon.
18. boolean cellIsRightBorder(int cellId) : prend en argument un entier cellId, teste si la cellule correspondant à l'identifiant cellId est sur une bordure de droite, retourne vrai si c'est le cas et faux sinon.
19. boolean cellIsBottomBorder(int cellId) : prend en argument un entier cellId, teste si la cellule correspondant à l'identifiant cellId est sur une bordure inférieure, retourne vrai si c'est le cas et faux sinon.
20. Cell getNeighbor(int dir, int cellId, Cell[][] halfFloor) : prend en argument deux entiers dir et cellId et un demi-étage halfFloor, retourne la cellule voisine dans la direction correspondant à la variable dir de la cellule correspondant à l'identifiant cellId dans le demi-étage halfFloor.
21. boolean destroyWall(int dir, int cellId, Cell[][] halfFloor) : prend en argument deux entiers dir et cellId et un demi-étage halfFloor, teste si le mur correspondant à la variable dir de la cellule correspondant à l'identifiant cellId est destructible, si oui



alors il est détruit en utilisant la fonction `destroyWall` de `Cell`, retourne la valeur retournée par `destroyWall`.

22. `boolean cellIsHole(int cellId)` : prend en argument un entier `cellId`, teste si la variable `cellId` est dans la variable interne `holes`, retourne vrai si c'est le cas et faux sinon.
23. `void print()` : ne prend pas d'argument, affiche les deux demi-étages `top` et `bottom` sur la sortie standard en utilisant la fonction `print` dans `Cell`.

## A.4 Maze

### A.4.1 Variables

1. `firstFloor = Floor` : représente le premier étage du labyrinthe.
2. `height = int` : représente le nombre d'étages du labyrinthe.
3. `generated = boolean` : représente si le labyrinthe a été généré par l'algorithme ou non.

### A.4.2 Fonctions

1. `Floor getFirstFloor()` : ne prend pas d'argument, retourne la variable interne `firstFloor`.
2. `int getHeight()` : ne prend pas d'argument, retourne la variable interne `height`.
3. `void increaseHeight()` : ne prend pas d'argument, incrémente la variable interne `height`, ne retourne rien.
4. `boolean isGenerated()` : ne prend pas d'argument, retourne la variable interne `generated`.
5. `void makeGenerated()` : ne prend pas d'argument, met la variable interne `generated` à vrai, ne retourne rien.
6. `Floor getLastFloor()` : ne prend pas d'argument, parcourt les étages un par un en utilisant les variables internes aux étages `next`, retourne le dernier étage obtenu.
7. `void addFloor()` : ne prend pas d'argument, initialise la variable interne `next` du dernier étage obtenu grâce à la fonction interne `getLastFloor` par un étage, ne retourne rien.
8. `void addFloors(int nbFloors)` : prend en argument un entier `nbFloors`, utilise la fonction interne `addFloor` pour ajouter `nbFloors`, ne retourne rien.
9. `int[][] getCoordHoles()` : ne prend pas d'argument, retourne une liste de toutes les listes de points de passage des étages.
10. `void print()` : ne prend pas d'argument, affiche les étages un par un sur la sortie standard en utilisant la fonction `print` dans `Floor`.

## A.5 fusion

### A.5.1 Variables

1. `hard = boolean` : représente si le niveau de difficulté est dur ou non.

### A.5.2 Fonctions

1. `void generateMaze(Maze maze)` : prend en argument un `Maze maze`, utilise la fonction interne `generateFloor` sur tous les étages de `maze` pour générer le labyrinthe, ne retourne rien.

2. `void generateFloor(Floor floor)` : prend en argument un `Floor floor`, si la variable interne `hard` est vrai alors utilise la fonction interne `generateFloorHard` sinon la fonction interne `generateFloorMedium` pour générer l'étage `floor`, ne retourne rien.
3. `void generateFloorMedium(Floor floor)` : prend en argument un `Floor floor`, utilise la fonction interne `generateHalfFloorMedium` sur les variables internes `top` et `bottom` de `floor` pour générer l'étage `floor`, ne retourne rien.
4. `void generateFloorHard(Floor floor)` : n'a pas été implémentée, doit être similaire à `generateFloorMedium` mais avec une difficulté supérieure.
5. `generateHalfFloorMedium(Cell[][] halfFloor, Floor floor)` : prend en argument un demi-étage `halfFloor` et un `Floor floor`, utilise l'algorithme de fusion aléatoire de chemins décrit dans le rapport pour générer le demi-étage `halfFloor` dans l'étage `Floor`, ne retourne rien.
6. `void makeSet(Cell[][] halfFloor, Floor floor)` : prend en argument un demi-étage `halfFloor` et un `Floor floor`, initialise les variables internes `parent` et `rank` des cellules du demi-étage `halfFloor` dans l'étage `floor` avec les fonctions internes `setParent` et `setRank` dans `Cell`, ne retourne rien.
7. `Cell find(Cell cell)` : prend en argument un `Cell cell`, teste si la variable interne `parent` de `cell` est `cell`, retourne `cell` si c'est le cas et la valeur retournée par la fonction interne `find` sur la variable interne `parent` de `cell` sinon.
8. `void union(Cell cellA, Cell cellB)` : prend en argument deux `Cell cellA` et `cellB`, utilise la fonction interne `find` pour trouver les représentants de `cellA` et `cellB` puis teste si ces représentants sont les mêmes, si oui ne fait rien et sinon modifie le représentant de l'un des deux grâce à la fonction interne `setParent` de `Cell`.
9. `int[] getPossibleHoles(int size)` : prend en argument un entier `size`, retourne la liste des identifiants des cellules qui peuvent contenir un trou dans un étage de taille `size`.
10. `void createHole(Floor floor)` : prend en argument un `Floor floor`, utilise la fonction interne `getPossibleHoles` pour récupérer la liste des identifiants des cellules pouvant avoir un trou dans `floor` et ajoute un trou à la variable interne `holes` dans `floor`, ne retourne rien.

## A.6 Converter

### A.6.1 Fonctions

1. `ndArray[] halfFloorToMatrix(Cell[][] halfFloor, int size)` : prend en argument un demi-étage `halfFloor` et un entier `size`, parcourt le demi-étage `halfFloor` et crée deux matrices contenant les murs l'une verticaux l'autre horizontaux, retourne une liste contenant ces deux matrices.
2. `ndArray[] floorToMatrix(Floor floor)` : prend en argument un `Floor floor`, utilise la fonction interne `halfFloorToMatrix` aux deux demi-étages de `floor`, retourne la liste des valeurs retournées par la fonction interne `halfFloorToMatrix` sur chaque demi-étage.
3. `ndarray[][] mazeToMatrix(Maze maze)` : prend en argument un `Maze maze`, utilise la fonction interne `floorToMatrix` aux étages de `maze`, retourne la liste des valeurs retournées par la fonction interne `floorToMatrix` sur chaque étage.