

# Projet final et SAÉ : quadtree

## Initiation au développement

### BUT informatique, première année

Ce projet vise à matérialiser les déplacements d'un personnage sur un terrain potentiellement infini, ou, plus précisément, engendré au fur et à mesure de son exploration. La figure 1 montre un aperçu de l'aspect du projet tel qu'il vous sera fourni au départ.

Le terrain sera stocké en mémoire à l'aide d'une structure de données appelée quadtree (d'où le titre du projet), ou arbre quaternaire, pour laquelle vous devrez développer une bibliothèque. Il s'agit d'une structure de données classique qui trouve des applications dans différents domaines de l'informatique, notamment liés à l'imagerie. Nous l'utiliserons pour sa capacité à stocker la structure d'une (très) grande image de manière compacte.

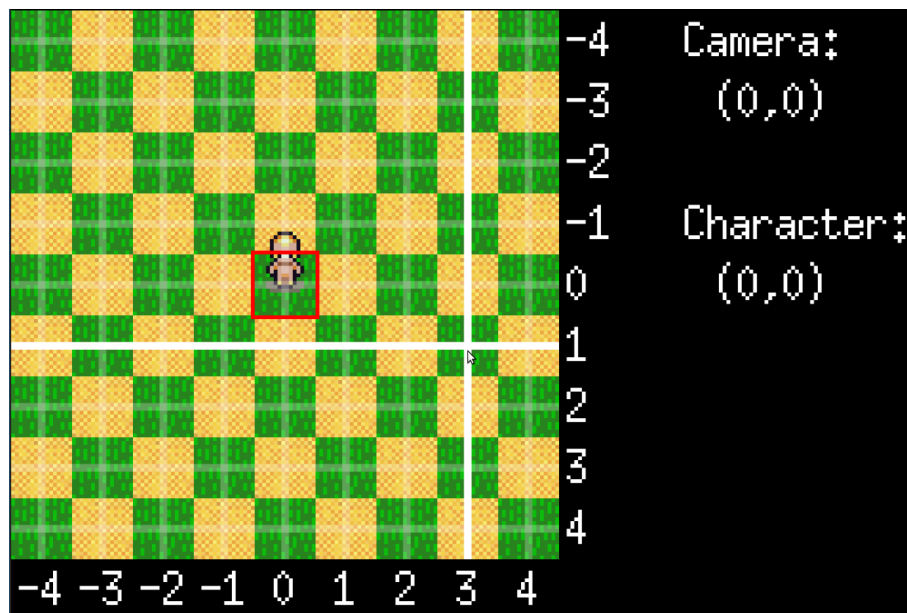


FIGURE 1 – Aperçu du projet, en mode debug, tel qu'il vous est fourni au départ. On peut observer un personnage situé aux coordonnées (0,0) sur un terrain carré de 9 cases de côtés. Le carré rouge matérialise la position de la caméra, elle aussi située aux coordonnées (0,0). Sur les côtés du terrain les coordonnées sont rappelées. La souris repère actuellement la case de coordonnées (3,1).

## 1 Évaluation et dates

Ce projet comptera à la fois pour le cours d'introduction au développement et pour la SAÉ implémentation d'un besoin client.

### 1.1 Dates

- C'est le travail de la partie 5 ci-dessous qui comptera pour le cours d'introduction au développement. Vous devrez le rendre au plus tard le **vendredi 20 décembre 2024** (semaine 51).
- C'est le travail de la partie 6 ci-dessous qui comptera pour la SAÉ implémentation d'un besoin client. Vous devrez le rendre en semaine 3 à une date qui sera précisée ultérieurement.

- Vous évalueriez aussi le code d'un autre groupe en semaine 3 juste après votre rendu.
- Il y aura enfin un contrôle sur table en fin de semaine 3, à une date qui sera déterminée plus tard. Ce test portera sur la compréhension générale du projet, vous pourrez avoir des questions sur mon code, votre code, les algorithmes mis en œuvre, les améliorations possibles, etc.

## 1.2 Critères d'évaluation

Pour les deux parties, seront évalués (par ordre d'importance) : le bon fonctionnement du code, le choix et la qualité des cas de tests, la qualité du code (choix judicieux de noms de variables et fonctions, organisation claire des fichiers, documentation bien ciblée, etc), l'efficacité du code.

Pour la SAÉ, le nombre d'extensions réalisées et leur difficulté sera aussi un critère important d'évaluation (il ne s'agit pas de réaliser toutes les extensions, bien sûr, mais d'en réaliser plusieurs et pas seulement les plus simples), de même qu'un recul important sur le projet (une compréhension précise de tout le fonctionnement du code, y compris celui qui vous a été fourni). Ce dernier point sera notamment évalué lors du contrôle sur table.

Notez bien qu'il est recommandé d'être présents à toutes les séances de SAÉ qui seront prévues dans l'emploi du temps : elles peuvent être l'occasion pour les encadrants de venir vous questionner sur votre avancement (ce qui pourra compter pour l'évaluation) mais aussi, et surtout, elle seront l'occasion pour vous de vous faire aider et ainsi de ne jamais rester bloqués trop longtemps sur un point de difficulté.

Pour que l'évaluation puisse être individualisée, vous devrez être en mesure d'indiquer précisément quelles parties du projet ont été développées par quel étudiant. Nous nous réservons aussi le droit d'analyser vos historiques sur Gitlab. Chaque membre d'un groupe doit contribuer significativement au projet en écrivant du code et en le soumettant sur Gitlab.

## 1.3 Rendre son travail

Vous travaillerez par groupes de 2 (avec un groupe de 3 par groupe de TD si nécessaire).

Pour rendre votre travail vous utiliserez obligatoirement un dépôt sur le Gitlab de l'Université (accessible à l'adresse <https://gitlab.univ-nantes.fr/>) à l'aide duquel vous travaillerez tout au long du projet.

Au moment de chaque rendu vous créerez une *release* sur votre dépôt (dans le menu de gauche d'un dépôt, faire *Deployments* puis *Releases* et suivre les instructions). C'est le contenu de cette *release* qui sera évalué.

Pensez bien à régulièrement ajouter votre travail au serveur (utiliser très souvent la commande `git commit` et, à chaque fin de session de travail, la commande `git push`), en l'identifiant bien avec votre nom et votre prénom et en mettant des messages explicites accompagnant vos soumissions (`commit`). Ceci nous permettra de voir à quel rythme vous avez travaillé et quelle a été la contribution de chaque membre du groupe.

## 2 Organisation des sources du projet

Des sources vous sont fournies avec le projet pour vous donner un point de départ et vous permettre de rentrer directement dans le vif du sujet sans avoir à mettre en place des solutions techniques (notamment pour l'affichage du personnage et du terrain) avant de commencer. Vous devez respecter leur organisation tout au long du projet et vous ne devez pas la changer (si vous avez vraiment besoin de le faire ça doit être avec l'accord d'un enseignant et il faudra pouvoir le justifier).

### 2.1 Dossier principal

Le dossier principal du projet contient les fichiers `go.mod` et `go.sum` ainsi que les dossiers correspondant aux différents paquets (et un dossier `floor-files` qui contient des exemples de fichiers représentant des terrains) :

- `assets` : images et code pour les importer (2.2)
- `camera` : gestion de la caméra (2.3)
- `character` : gestion du personnage (2.4)
- `cmd` : construction de l'exécutable (2.5)
- `configuration` : lecture des fichiers de configuration (2.6)

- floor : gestion du terrain (2.7)
- game : implantation de l'interface `ebiten.Game` (2.8)
- quadtree : bibliothèque pour les quadtree (2.9)

Si vous regardez le contenu du fichier `go.mod` vous noterez que le projet utilise la bibliothèque Ebitengine ([github.com/hajimehoshi/ebiten/v2](https://github.com/hajimehoshi/ebiten/v2)). Celle-ci fournit des méthodes et fonctions utiles pour créer un jeu vidéo : affichage d'images à l'écran, détection des interactions de l'utilisateur avec les touches du clavier, etc, et surtout cadencage des calculs à 60 images par seconde. Ceci est un peu plus détaillé dans la partie 3.

## 2.2 Paquet *assets*

Le dossier *assets* contient les images utilisées pour le projet (une image pour le personnage : `character.png`, et une pour le terrain : `floor.png`) ainsi que le code du paquet *assets* qui permet de charger en mémoire ces images pour les utiliser ensuite (fonction `Load`).

## 2.3 Paquet *camera*

Le paquet *camera* permet de gérer une caméra abstraite, qui représente en fait le centre du terrain affiché. Pour l'instant, deux versions de la caméra sont fournies : une caméra statique, qui reste tout le temps aux mêmes coordonnées, et une caméra qui suit le personnage.

Cette caméra est mise-à-jour 60 fois par seconde par la méthode `Update`.

## 2.4 Paquet *character*

Le paquet *character* permet de gérer le personnage : mise-à-jour de sa position en fonction des touches utilisées par l'utilisateur, affichage à l'écran en fonction de la position de la caméra.

La mise-à-jour est réalisée par la méthode `Update` et l'affichage par la méthodes `Draw`. Les données utiles à ces deux méthodes sont décrites dans le type `Character` (position du personnage, orientation, etc).

## 2.5 Dossier *cmd*

Le dossier *cmd* contient le code utilisé pour produire l'exécutable du projet (le paquet *main*, donc). C'est dans ce dossier qu'il faut donc appeler `go build` pour pouvoir faire fonctionner le projet.

Ce dossier contient un fichier `config.json` qui permet de paramétrer le projet au lancement. Ainsi, vous pourrez tester plusieurs configurations différentes sans avoir à recompiler. Essayez de rendre votre résultat final le plus configurable possible : cela simplifiera la vie de vos encadrants (et la votre lors du développement).

Le programme peut fonctionner dans deux modes :

mode debug : les coordonnées du personnage et de la caméra sont affichées à l'écran, une grille est visible par dessus le terrain (et on peut passer sa souris dessus pour mettre en surbrillance les coordonnées d'une case), un carré rouge matérialise la caméra. Ce mode s'active lorsque le champs `DebugMode` de la configuration est à `true`. On peut aussi, en cours d'exécution, l'activer/le désactiver en appuyant sur la touche `d` du clavier.

mode normal : seuls le personnage et le terrain sont affichés.

Durant tout le projet, si vous ajoutez des informations de debug (par exemple avec la bibliothèque `log`), laissez-les dans votre code en les mettant dans une conditionnelle pour les afficher seulement en mode debug. Ceci permettra à la personne qui corrigera votre projet de voir plus en détails ce qui se passe si elle le souhaite, sans avoir à ajouter elle-même du code Go.

## 2.6 Paquet *configuration*

Le paquet *configuration* offre une fonction `Load` qui permet de lire un fichier de configuration (comme celui disponible dans le dossier *cmd*) et de charger son contenu dans une variable de type `Configuration`. La variable en question est une variable globale définie au niveau de ce paquet : votre projet ne pourra charger qu'une seule configuration à la fois.

Pour enrichir le fichier de configuration il suffit d'ajouter des champs au type `Configuration`.

## 2.7 Paquet *floor*

Le paquet *floor* offre une méthode `Update`, permettant de mettre à jour le terrain en fonction de la caméra (c'est-à-dire de calculer la zone du terrain qui est visible à l'écran pour une position donnée de la caméra) et une méthode `Draw`, permettant d'afficher le terrain à l'écran.

Un terrain est un grand tableau de cases, chacune représentant un type de sol (herbe, désert, eau, etc). Chaque case possède des coordonnées absolues (ses coordonnées dans le grand tableau) et des coordonnées relatives (ses coordonnées à l'écran).

## 2.8 Paquet *game*

Le paquet *game* implante l'interface `ebiten.Game`, c'est ce qui permet d'utiliser la bibliothèque Ebitengine pour faire fonctionner le projet correctement. Plus de détails sont donnés dans la partie 3.

## 2.9 Paquet *quadtree*

Le paquet *quadtree* servira à implanter une bibliothèque fournissant une structure de données pour des arbres quaternaires. Pour l'instant (une première version de) la structure de données est fournie mais les fonctions et méthodes ne sont pas codées.

# 3 La bibliothèque Ebitengine

Attention, pour pouvoir utiliser la bibliothèque Ebitengine vous devez installer les paquets suivants sur votre machine virtuelle Ubuntu : `libgl1-mesa-dev` et `xorg-dev`

Ebitengine<sup>1</sup> est une bibliothèque de développement de jeux vidéos en 2D (un moteur de jeux, *game engine*) qui possède une API (l'ensemble des méthodes et fonctions fournies) simple et qui permet de développer pour de nombreuses plateformes (Windows, Linux, MacOS, Android, iOS, Switch).

Cette bibliothèque se charge en particulier de simplifier trois aspects fondamentaux de la création d'un jeu : afficher des images à l'écran (partie 3.1), détecter les entrées du joueur (partie 3.3), cadencer les calculs (partie 3.2).

L'API se base sur une interface `ebiten.Game` qui indique les trois méthodes qui doivent être mises en œuvre par un jeu : `Update` (pour mettre à jour l'état du jeu à chaque 1/60 de seconde), `Draw` (pour afficher des choses à l'écran) et `Layout` (pour définir la taille de la zone d'affichage en fonction de la taille de la fenêtre dans laquelle cet affichage a lieu).

### 3.1 Afficher des images à l'écran

Pour simplifier l'affichage, Ebitengine propose de tout représenter par des images (`ebiten.Image`) et de fournir des fonctions pour dessiner des images sur d'autres images (éventuellement en les transformant, figure 2a). En particulier, l'écran est représenté par une image qui est rendue accessible dans la méthode `Draw` nécessaire pour qu'une structure implante l'interface `ebiten.Game`.

En interne (c'est transparent quand on utilise la bibliothèque) Ebitengine se charge d'optimiser la gestion des images en les chargeant efficacement en mémoire (dans ce qu'on appelle *Texture Atlas*) et en organisant les opérations sur les images de manière à optimiser l'utilisation de la carte graphique (figure 2b).

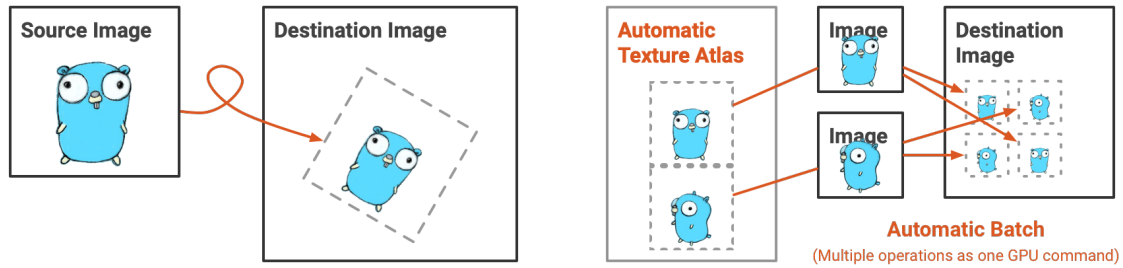
### 3.2 Cadencer les calculs

Pour qu'un jeu s'exécute de manière fluide, il est en général préférable de fonctionner par étapes de calcul ayant lieu régulièrement dans le temps : cela simplifie le contrôle des vitesses de déplacement relatives des éléments du jeu (sans avoir à mesurer des temps puisqu'on sait exactement combien de fois les calculs seront faits par seconde) et l'animation des éléments visuels.

Ebitengine gère cela pour nous. Quand on utilise la fonction `ebiten.RunGame` de Ebitengine sur une structure qui implante l'interface `ebiten.Game` (c'est-à-dire, qui dispose des méthodes `Update`, `Draw` et `Layout`) ceci démarre

---

1. <https://ebitenengine.org>



(a) Gestion simple (source <https://ebiten.org>) (b) Optimisation (source <https://ebiten.org>)

FIGURE 2 – Gestion des images par Ebitengine

une boucle qui va, à chaque tour, exécuter **Update** puis **Draw** et ensuite attendre le temps qu'il faut pour que le tour de boucle dure exactement 1/60 de seconde (figure 3). On peut remarquer que cette stratégie a la conséquence suivante : si le temps mis pour exécuter **Update** puis **Draw** est supérieur à 1/60 de seconde, alors le jeu a des ralentissements.

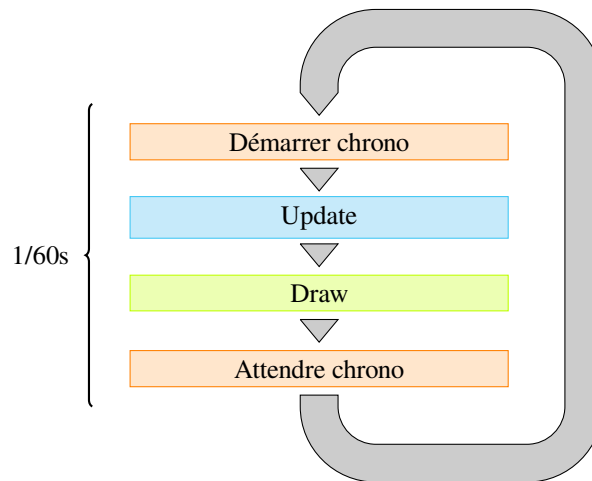


FIGURE 3 – Principe de la fonction `ebiten.RunGame`

### 3.3 Détecter les entrées du joueur

Ebitengine dispose d'un grand tableau d'entiers avec une case pour chaque touche susceptible d'être pressée. À chaque pas de temps (chaque 1/60 de seconde) ce tableau est automatiquement mis-à-jour de la façon suivante :

- si une touche est pressée, la valeur de la case correspondante est augmentée de 1,
- si une touche n'est pas pressée, la valeur de la case correspondante est mise à 0.

Ceci permet de savoir, à chaque pas de temps, tous les boutons qui ont été pressés (ou sont en train d'être pressés) depuis le pas de temps précédent.

En pratique, ce tableau n'est pas visible pour les utilisateurs : des fonctions qui indiquent si une touche est pressée ou non (en lisant le tableau) sont fournies à la place.

Quand vous utiliserez la bibliothèque Ebitengine vous pourrez vous reporter à la documentation, accessible à l'adresse <https://pkg.go.dev/github.com/hajimehoshi/ebiten/v2>. Faites bien attention au fait que nous utilisons la v2 de la bibliothèque (en cherchant dans un moteur de recherche on peut tomber sur la documentation de la v1, qui a quelques différences fondamentales.)

## 4 Informations générales sur les quadtree

Un quadtree, ou arbre quaternaire en français, est un type d'arbre où chaque nœud qui n'est pas une feuille possède exactement quatre enfants. Dans ce projet, on utilisera des quadtrees particuliers pour représenter le terrain de manière compacte (ces quadtrees sont habituellement utilisés pour stocker les couleurs des pixels d'une image, ce qui revient au même que, dans notre cas, de stocker les types de sol (herbe, sable, etc) des cases d'un terrain).

La figure 4 montre le principe de la représentation d'un terrain par un arbre. Pour simplifier, cet exemple est basé sur un terrain carré dont le côté contient un nombre de cases qui est une puissance de 2.

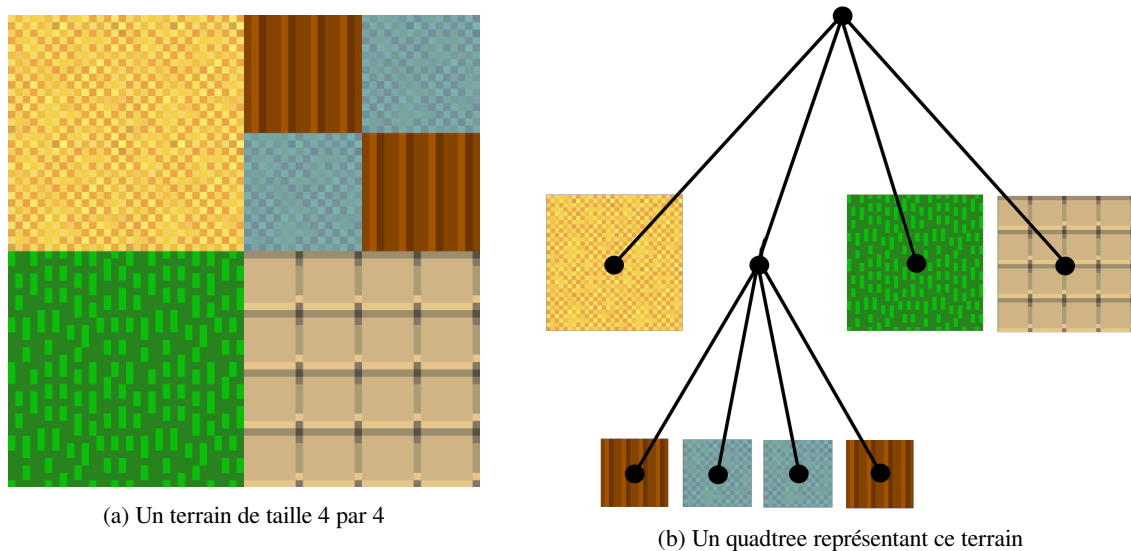


FIGURE 4 – Représentation d'un terrain par un arbre quaternaire

L'idée est que le terrain est découpé en quatre zones, chacune d'entre elles étant représentée par un nœud de l'arbre. Ici, comme le terrain a huit cases de côté on peut faire quatre zones carrées de deux cases de côté (mais, si le terrain était rectangulaire, on aurait pu faire un autre découpage, l'important est d'avoir quatre zones qui ne se chevauchent pas et qui couvrent tout le terrain). Trois des zones obtenues ne contiennent qu'un seul type de sol (celle en haut à gauche, celle en bas à gauche, celle en bas à droite), elles sont donc représentées par des feuilles de l'arbre (qui contiennent l'information de la taille de ces zones et du type de sol qui les constitue). La quatrième zone est constituée de plusieurs types de sol, elle est donc redécoupée en quatre : le nœud qui la représente a quatre enfants.

Pour un terrain plus grand et plus complexe, le même principe se serait appliqué mais cela aurait engendré un arbre plus profond. On peut représenter n'importe quel terrain rectangulaire avec cette méthode.

## 5 Premiers travaux : lecture de fichiers et quadtrees

Pour découvrir le projet, vous commencerez (partie 5.1) par coder une fonction qui permet de lire un fichier représentant un terrain et de le stocker en mémoire sous la forme d'un tableau pour ensuite pouvoir afficher à l'écran une zone de ce terrain (en fonction de la position de la caméra). Ensuite, vous remplacerez le stockage dans un tableau par un stockage sous forme de quadtree (partie 5.2). Ceci impliquera de développer une bibliothèque pour les quadtrees.

### 5.1 Récupération du terrain depuis un fichier

Votre premier travail est de coder la fonction `readFloorFromFile` et la fonction `updateFromFileFloor` du paquet `floor`, et de tester leur fonctionnement (en faisant des essais d'exécution du programme avec une configuration adaptée, et en écrivant des tests automatisés pour le paquet `floor`). Ces fonctions sont utilisées lorsque le champs `FloorKind` du fichier de configuration vaut 1.

**Fichiers représentant des terrains.** Les fichiers représentant des terrains utilisent la syntaxe suivante : chaque ligne du fichier représente une ligne du terrain, sur une ligne, chaque caractère représente le type de sol d'une case du terrain. Comme les types de sol sont numérotés de 0 à 4, on utilisera les mêmes numéros pour les représenter dans les fichiers. La figure 5 donne en exemple le contenu du fichier représentant le terrain de la figure 4a.

```
1134
1143
0022
0022
```

FIGURE 5 – Contenu du fichier représentant le terrain de la figure 4a

**readFloorFromFile.** Vous trouverez cette fonction dans le fichier `init.go` du dossier `floor`. Cette fonction lit un fichier dont le chemin est donné en paramètre et retourne le terrain correspondant sous la forme d'un tableau d'entiers. C'est à vous de créer le tableau aux bonnes dimensions (qui doivent être exactement celles du terrain représenté par votre fichier, déterminées à l'exécution en fonction du fichier lu).

**updateFromFileFloor.** Vous trouverez cette fonction dans le fichier `update.go` du dossier `floor`. Cette fonction fixe la valeur du champs `content` d'une variable de type `Floor` à partir du contenu du champs `fullContent` et de la position de la case en haut à gauche de `content` (obtenue en fonction de la position de la caméra). La position de cette case est donnée de manière absolue, c'est-à-dire par ses coordonnées dans `fullContent`. Il faut calculer `content`, qui fait exactement la taille de l'écran (largeur `configuration.Global.NumTileX`, hauteur `configuration.Global.NumTileY`) de telle sorte que la caméra soit placée au milieu de l'écran (c'est-à-dire que la case en haut à gauche dont vous avez les coordonnées soit bien en haut à gauche de l'écran). Il faut donc faire une conversion des coordonnées absolues (dans `fullContent`) aux coordonnées relatives (dans `content`).

La figure 6 donne un exemple de la façon dont `content` s'obtient à partir de `fullContent` et de la position de la caméra.

0	0	0	0	0	1	1	1	1	1	0
0	0	0	0	0	1	1	1	1	1	1
0	0	0	0	0	1	1	1	1	1	2
0	0	0	0	0	1	1	1	1	1	3
0	0	0	0	0	1	1	1	1	1	4
2	2	2	2	2	2	3	3	3	3	5
2	2	2	2	2	2	3	3	3	3	6
2	2	2	2	2	2	3	3	3	3	7
2	2	2	2	2	2	3	3	3	3	8
2	2	2	2	2	2	3	3	3	3	9
0	1	2	3	4	5	6	7	8	9	

0	0	1	1	1	0
2	2	3	3	3	1
2	2	3	3	3	2
2	2	3	3	3	3
2	2	3	3	3	4
0	1	2	3	4	

(a) `fullContent` avec la caméra en (5, 6) (représentée en rouge) et la case en haut à gauche de l'écran en (3, 4) (représentée en vert). Une zone de cinq cases par cinq est aussi dessinée autour bien placée. Ceci est obtenu à partir du tableau `fullContent` de la caméra (représentée en bleu, c'est la taille de ce qui doit être affiché à l'écran). Les coordonnées sont les coordonnées absolues. Les coordonnées sont les coordonnées relatives.

FIGURE 6 – Lien entre `fullContent`, `content` et la position de la caméra.

Il est possible que la zone visible à l'écran (`content`) dépasse du terrain (par exemple quand la caméra est en (0, 0)). Dans ce cas, il faudra compléter les cases vides de `content` avec la valeur -1 (qui est utilisée pour indiquer une absence de type de sol et qui bloque le personnage).

### Quelques conseils.

- La version initiale de `updateFromFileFloor` vous permet d'afficher à l'écran une copie de `fullContent` (à condition que l'écran soit assez grand, ce que vous pouvez régler dans la configuration). Ceci vous permet de tester votre fonction `readFloorFromFile` avant de coder `updateFromFileFloor`.
- Ne vous lancez pas tout de suite dans la programmation : commencez par tester le programme, essayez de comprendre comment fonctionne le fichier de configuration, lisez un peu le code fourni pour avoir une idée plus précise de l'architecture et du fonctionnement du projet.
- Après avoir codé chacune des deux fonctions demandées, testez avec les fichiers de terrain fournis. Écrivez aussi vos propres fichiers de terrain pour tester un peu plus largement. Testez enfin dans les différents modes de caméra et pour différentes tailles de la zone affichée à l'écran.
- Vous pouvez écrire (au moins en partie) les tests automatisés avant de commencer à coder les fonctions, ceci vous permettra de bien comprendre ce que vous avez à coder avant de vous lancer.
- Et bien sûr, pensez à modifier la configuration (mettre `FloorKind = 1`) pour indiquer au programme d'utiliser vos nouvelles fonctions et à tester différentes caméras quand vous aurez codé `updateFromFileFloor` (mettre `CameraMode = 1`).

**Attention**, les coordonnées à l'écran sont données en pixels à partir du coin supérieur gauche de la fenêtre : quand  $x$  augmente on se déplace vers la droite, quand  $y$  augmente on se déplace vers le bas. La même convention a été utilisée pour les coordonnées en cases sur le terrain.

## 5.2 Bibliothèque pour les quadrees

Une fois bien compris le fonctionnement des tableaux `fullContent` et `content`, on souhaite remplacer `fullContent` par un quadtree, pour des raisons d'efficacité du stockage de l'information. Pour cela, on va créer une bibliothèque pour les quadrees (paquet `quadtree`).

Une structure de données pour représenter ces quadrees est déjà proposée. Il reste à implanter la fonction `MakeFromArray` (responsable de stocker un tableau représentant un terrain sous la forme d'un quadtree) et la méthode `GetContent` (responsable de remplir le tableau `content` à partir d'un quadtree représentant tout le terrain).

Ces fonctions sont utilisées lorsque le champs `FloorKind` du fichier de configuration vaut 2.

**MakeFromArray.** Vous trouverez cette fonction dans le fichier `make.go` du dossier `quadtree`. Étant donné un tableau représentant un terrain retourné par la fonction `readFromFile` développée précédemment, cette fonction doit initialiser un quadtree représentant ce terrain. Pour cela, le terrain sera découpé en quatre zones rectangles de tailles aussi proches que possible (figure 7), chacune d'entre elles étant représentée par un nœud de l'arbre (qui est donc lui-même un quadtree). Notez bien que ceci invite à utiliser la récursivité.

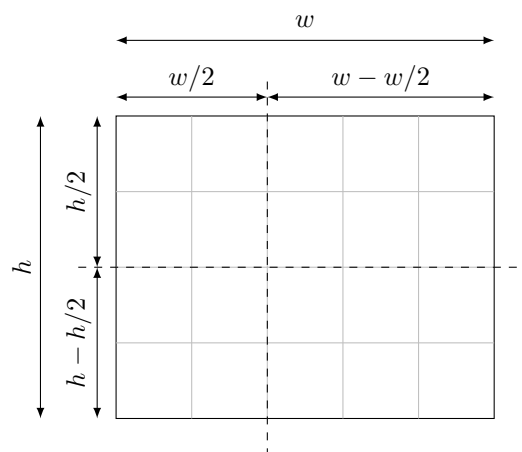


FIGURE 7 – Découpage d'un terrain en quatre zones rectangles de tailles aussi proches que possibles.

Il faudra bien faire attention à ce que chaque nœud de l'arbre contiennent toutes les informations demandées dans la structure de données `node`. De plus, il faudra s'assurer que, si l'intégralité de la zone de terrain représentée par



un nœud de l'arbre est constituée du même type de sol, ce nœud n'ait pas d'enfants : les quadrees doivent être aussi compactes que possible (on parle de réduction des quadrees).

Pour vous aider, le fichier `make_test.go` contient des tests pour vérifier le bon fonctionnement de votre fonction. Ces tests suggèrent de procéder au développement par étapes :

1. coder la fonction `MakeFromArray` dans le cas où le terrain est un carré de côté  $2^n$  avec  $n$  un entier et sans prendre en compte les réductions (testé par `TestMakeFromArrayPowerOfTwo`),
2. ajouter la prise en compte des terrains qui ne sont pas des carrés de côté  $2^n$  (testé par `TestMakeFromArrayAnyRectangle`),
3. ajouter la réduction des quadrees (testé par `TestMakeFromArrayReduction`).

**GetContent.** Vous trouverez cette fonction dans le fichier `get.go` du dossier `quadtree`. Cette fonction doit, étant donné un quadtree représentant tout le terrain et étant données les coordonnées du point en haut à gauche de la zone de terrain visible à l'écran, remplir un tableau `contentHolder` qui représente cette zone. Le tableau `contentHolder` est fourni à la bonne taille, il faut remplir toutes ses cases. Ce tableau est une copie du tableau `content` utilisé dans la partie précédente, modifier une de ses cases revient donc à modifier une case du tableau `content` (comme nous l'avons vu lors de l'apprentissage du Go).

Encore une fois, des tests vous sont fournis (dans le fichier `get_test.go`) pour guider votre développement :

1. `TestGetContentNoReduction` traite le cas des quadrees non réduits,
2. `TestGetContentReduction` traite le cas général.

#### Quelques conseils.

- Progresser par petites étapes en suivant les tests, ne faites pas tout d'un coup.
- Une fois codées l'étape 1 de `MakeFromArray` et l'étape 1 de `GetContent` vous serez en mesure de visualiser vos terrains dans le jeu. Un bon ordre pour le développement pourrait donc être de faire ces deux étapes, tester, puis l'étape 2 de `MakeFromArray`, tester. Enfin, vous pourrez ajouter l'étape 3 de `MakeFromArray` puis l'étape 2 de `GetContent`.
- N'hésitez pas à utiliser la méthode `GetAsText` du paquet `quadtree` pour visualiser vos quadrees sous forme textuelle et vérifier s'ils sont corrects.

## 6 La suite : extensions

La première partie du projet était très cadrée, cette deuxième partie est beaucoup plus libre. Vous devez coder quelques unes des extensions ci-dessous (elles sont à peu près classées par difficulté, de la plus simple à la plus compliquée, n'en faites pas que des simples, n'hésitez pas à interagir avec vos professeurs pour choisir vos extensions). Vous pouvez aussi proposer de nouvelles extensions, mais celles-ci doivent être validées par un professeur avant que vous commenciez à travailler dessus.

Chaque extension doit pouvoir être activée et désactivée en utilisant le fichier de configuration `config.json`. De même, si elles s'y prêtent, vos extensions doivent pouvoir être paramétrées dans le fichier `config.json`.

Les extensions ne doivent rien casser de ce qui existe déjà et doivent toutes être compatibles entre elles.

### 6.1 Génération aléatoire de terrain

On peut ajouter la possibilité, en plus de lire le terrain depuis un fichier, d'engendrer aléatoirement un terrain d'une taille donnée (de préférence sous forme d'un quadtree).

### 6.2 Enregistrement d'un terrain

À condition d'avoir fait une extension de génération de terrain on peut mettre en place les outils nécessaires pour, à partir d'un quadtree, reconstituer le contenu d'un fichier de terrain puis le sauvegarder pour pouvoir le réutiliser plus tard.

### 6.3 Animation des décors

Pour rendre l'environnement un peu plus attrayant, il est possible d'animer les décors (par exemple, de faire bouger l'eau). Pour cela vous pouvez vous inspirer de l'animation du personnage. Vous pouvez aussi utiliser la planche de tuiles complète d'où est extrait le fichier `floor.png` (le lien est dans le fichier de licence dans le dossier `assets`) pour trouver des animations déjà faites.

### 6.4 Téléporteurs

Si le terrain est grand, il peut être intéressant de pouvoir se déplacer rapidement d'un point à un autre. Dans cette extension on propose, au moyen d'une touche (par exemple `t`), de pouvoir placer un portail à l'emplacement actuel du joueur. Si on appuie plus tard une autre fois sur cette touche, cela placera un deuxième portail à l'emplacement actuel du joueur. Une fois les deux portails placés, en marchant sur l'un d'eux on se retrouvera à l'emplacement de l'autre. On peut permettre au joueur de replacer ensuite ses portails (par exemple en déplaçant le plus ancien à la position actuelle du joueur si on appuie sur la touche choisie).

### 6.5 Interdiction de marcher sur l'eau

On peut rendre certains types de sol bloquants : le personnage ne peut pas marcher dessus. Ça peut être le cas de l'eau par exemple. Attention, dans ce cas il faut s'assurer que le point de départ du personnage n'est pas sur une case de terrain correspondant à un type de sol bloquant.

### 6.6 Caméra bloquée aux bords du terrain

Pour l'instant, la caméra, lorsqu'elle n'est pas fixe, reste toujours au niveau du personnage. Si le personnage approche d'un bord du terrain, on voit donc une zone noire (où le terrain n'existe pas). On voudrait faire en sorte que, si le personnage approche trop du bord, la caméra se bloque juste avant le moment où on verrait cette zone noire, afin qu'on puisse voir uniquement du terrain à l'écran. Ceci demande que la taille du terrain soit supérieure à celle de l'écran (sinon il y a nécessairement du noir à l'écran).

### 6.7 Caméra plus précise

Actuellement, la caméra, lorsqu'elle n'est pas fixe, se déplace case par case (quand le personnage a terminé de faire son déplacement d'une case à une autre, la caméra le suit). Ceci n'est pas du tout fluide. Il pourrait être intéressant de modifier la caméra pour la rendre plus fluide en lui donnant une position qui soit un couple de flottants (ou au moins d'entiers qui représentent un pixel plutôt qu'une case) afin qu'elle puisse suivre exactement la position du personnage.

### 6.8 Caméra cinématographique

Actuellement, la caméra, lorsqu'elle n'est pas fixe, suit en permanence le personnage. Cela peut être intéressant de faire en sorte que la caméra ne se déplace que si le personnage se déplace beaucoup (elle reste fixe pour de petits mouvements et se met à se déplacer si le personnage menace de sortir de l'écran).

Cela peut aussi être plus esthétique de faire en sorte que la caméra ne démarre pas d'un coup quand le personnage se déplace mais augmente sa vitesse peu à peu jusqu'à rattraper le personnage. De même, quand le personnage s'arrête, il est possible de faire en sorte que la caméra ralentisse peu à peu plutôt que de s'arrêter d'un coup.

On trouve de nombreuses ressources en ligne sur les gestions de caméra. En voici par exemple une assez simple qui peut vous inspirer : <https://www.youtube.com/watch?v=01AHrMZ2Qvs>

### 6.9 Affichage du terrain amélioré

Lorsque une zone de terrain d'un certain type de sol est contenue dans une autre zone d'un autre type de sol (par exemple du sable au milieu d'une zone d'herbe) l'affichage actuel n'est pas très satisfaisant : on a des angles droits partout, ce qui ne fait pas très naturel. Dans cette extension, on propose d'améliorer cet affichage en s'inspirant de la figure 8. Pour cela, vous pourrez utiliser la planche de tuiles complète d'où est extrait le fichier `floor.png` (le lien est dans le fichier de licence dans le dossier `assets`).

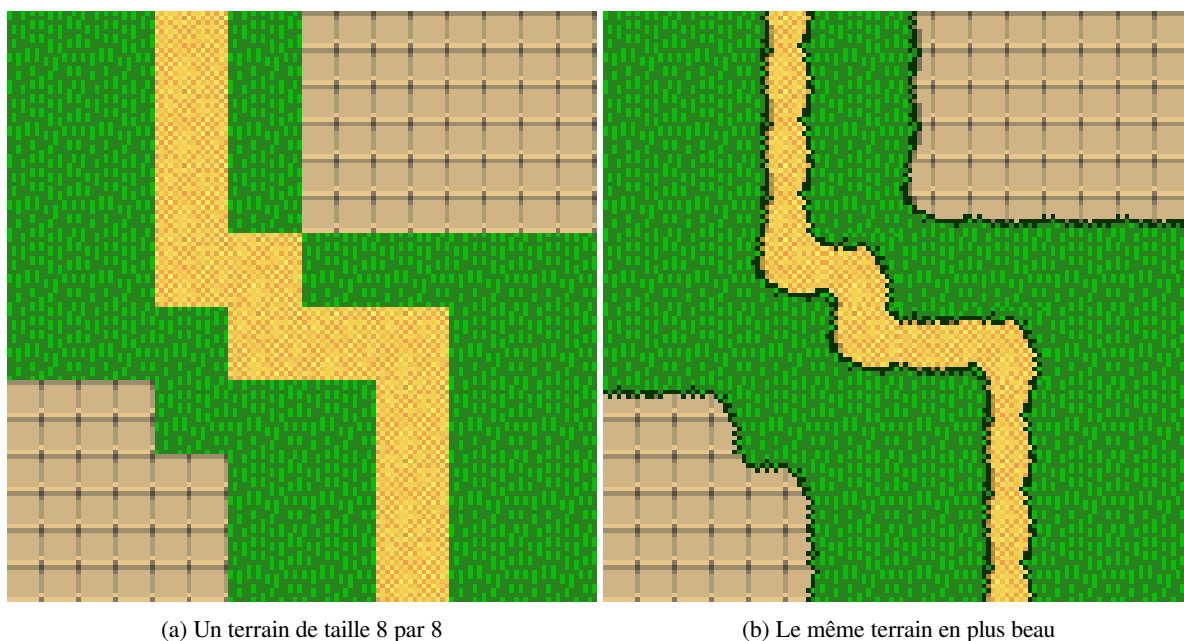


FIGURE 8 – Deux versions de l’affichage d’un même terrain

Vous pourrez remarquer (par exemple en vous renseignant sur internet) que, contrairement à ce qu’on pourrait penser au premier abord, il n’est pas nécessaire d’avoir  $2^8$  versions de chaque type de terrain mais seulement 15. Pour cela on pourra décaler d’une demi case en abscisse et en ordonnée la grille d’affichage par rapport à la grille du terrain.

## 6.10 Particules

On peut ajouter un petit système de particules (une nouvelle bibliothèque) pour simuler, par exemple, les effets des déplacements du personnage sur le terrain (traces de pas, poussière qui se soulève derrière lui, etc).

Un système de particules gère en permanence un ensemble de particules (des objets ayant une position, une vitesse et une image pour les afficher) en faisant, à chaque 1/60 de seconde, une mise-à-jour de chaque objet (la position est recalculée en ajoutant la vitesse à chaque coordonnée), une éventuelle création de nouveaux objets, puis un affichage de tous les objets (en utilisant leur position et l’image qui leur est associée).

## 6.11 La terre est ronde

On peut simuler le fait que le terrain est placé sur une sphère en faisant en sorte que, quand on arrive au bout du terrain d’un côté (par exemple en haut), on se retrouve au bout du terrain du côté opposé (dans l’exemple, en bas). Il faudra pour cela faire en sorte que le terrain s’affiche correctement (en changeant la façon dont le champs `content` du type `Floor` est mis-à-jour à chaque 1/60 de seconde) : il ne devrait plus jamais y avoir de zones noires visibles à l’écran.

## 6.12 Zoom

On souhaite pouvoir zoomer et dézoomer pour voir une plus ou moins grande zone de terrain.

## 6.13 Génération de terrain à la volée au fur et à mesure de l’exploration

On souhaite permettre d’explorer un terrain qui semble infini. Cependant, générer un terrain infini demanderait un temps infini et un espace de stockage infini : ce n’est donc pas possible. Pour simuler cela, on propose de générer le terrain au fur et à mesure de l’exploration : dès que le personnage va arriver dans une zone pas encore générée, celle-ci est construite sans qu’on s’en rende compte visuellement. Le terrain doit rester stocké dans un quadtree.

Pour cela, on propose de fonctionner de la manière qui est représentée dans l'exemple de la figure 9.

Initialement, on génère une zone d'une taille fixée et on place le personnage dedans (la zone A sur la figure 9a).

Quand le personnage se déplace, s'il sort de la zone générée (plus précisément, si une case qui ne fait pas partie de cette zone doit être affichée à l'écran) on a besoin d'agrandir celle-ci. Pour cela, on va utiliser la structure des quadtree. La zone déjà existante (A dans l'exemple) est représentée par un quadtree. La racine de celui-ci va devenir l'un des quatre enfants d'une nouvelle racine pour le quadtree. Les trois autres enfants (B, C et D sur la figure 9b) de cette racine représenteront trois autres zones du terrain, de la même taille que A et placées côte à côte. On pourra noter que seule la zone où le personnage vient de rentrer (C sur la figure 9b) a besoin d'être générée, les autres (B et D sur la figure 9b) peuvent être représentées par un simple nœud sans enfants dont le type de sol est une valeur spéciale (par exemple un nombre négatif) indiquant qu'elles ne sont pas encore générées.

Au fur et à mesure des déplacements du personnage on continue à générer les zones dans lesquelles le personnage entre (comme D sur la figure 9c). Et on agrandi le quadtree comme à l'étape précédente seulement quand c'est nécessaire, c'est-à-dire quand le personnage entre dans une zone qui ne correspond à aucun nœud (zone G sur la figure 9c).

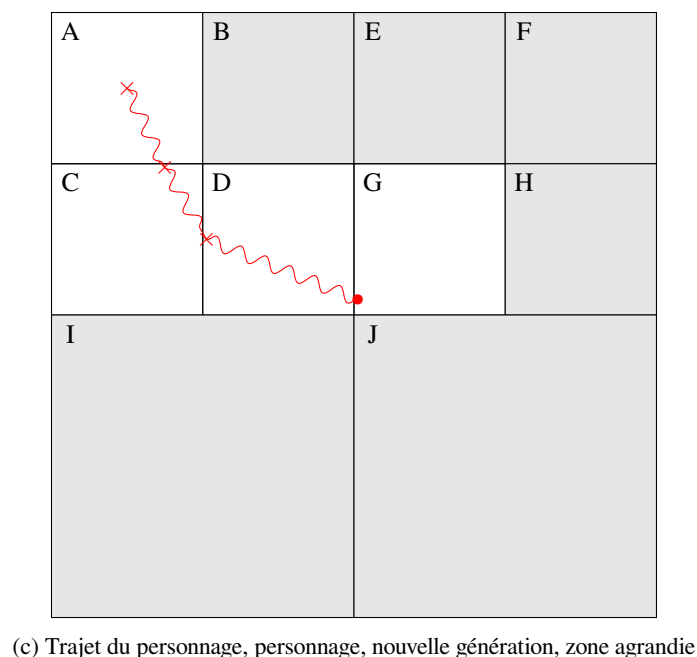
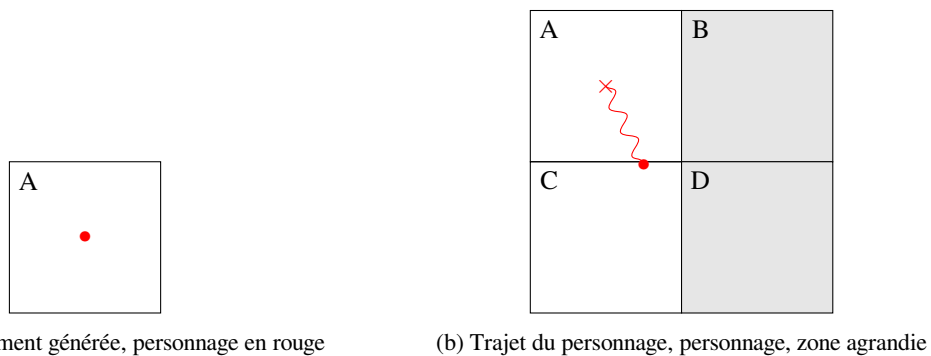


FIGURE 9 – Génération à la volée du terrain en fonction des déplacements du personnage.

Il faut bien faire attention à ce que les quadtrees restent corrects au fil de la génération (point en haut à gauche de chaque zone bien indiqué dans le nœud correspondant, idem pour la taille de chaque zone, etc).

## **6.14 Optimisation des quadtree**

Pour l'instant, lorsqu'on lit un fichier, on crée un quadtree arbitraire (i.e. correspondant à un découpage possible de l'image en rectangles ne contenant chacun qu'un seul type de sol). Dans cet extension on propose d'essayer d'analyser le fichier avant de le transformer en quadtree de manière à choisir les rectangles de telle sorte que le quadtree produit soit le plus compact possible.

## **6.15 Génération de terrains sans blocages**

Ceci nécessite d'avoir fait les extensions génération aléatoire de terrain et interdiction de marcher sur l'eau. De plus, ça ne devient une extension vraiment compliquée que si elle est combinée avec la génération du terrain à la volée au fur et à mesure de l'exploration.

L'idée est de s'assurer, lorsqu'on génère un terrain, que chaque case non bloquante de celui-ci est atteignable par le personnage (éventuellement en faisant un grand détours, qui n'a peut-être pas encore été généré dans le cas de la génération à la volée).