IFT1015 – Travail pratique #1 – 2018-10-16

DÉMINEUR

Marc Feeley

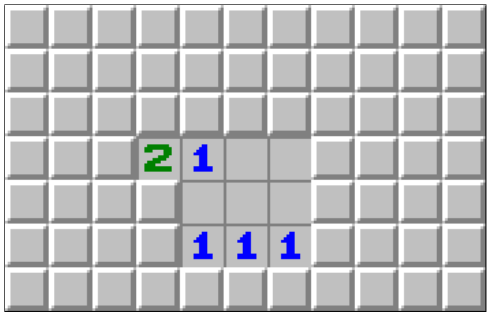
Le TP1 a pour but de vous faire pratiquer les concepts suivants : les boucles, les tableaux, les structures, les fonctions et les tests unitaires.

Vous avez à coder et tester un petit jeu graphique qui est une variante simplifiée du jeu de **démineur** (`https://fr.wikipedia.org/wiki/Démineur_(jeu)`). Le développement doit se faire en JavaScript à l'aide de l'environnement codeBoot et vous devez vous limiter aux aspects du langage que nous avons vus dans le cours jusqu'à la partie sur les structures et tableaux inclusivement.

---

1 Introduction

Le jeu se joue sur une grille de tuiles. Chaque tuile est une petite image de 16x16 pixels. Voici un exemple de ce qui pourrait être affiché au milieu du jeu après avoir joué quelques coups :



Avant d'expliquer le fonctionnement du jeu il faut comprendre comment l'affichage des tuiles se fait.

Sur le plan matériel, l'affichage d'images (et aussi de texte) à l'écran d'un ordinateur se fait en contrôlant le contenu d'une grille rectangulaire de pixels carrés. La grille a une largeur et hauteur qui dépend de la résolution de l'écran (par exemple 1600 colonnes par 1200 rangées). Chaque pixel a une coordonnée X-Y qui indique sa position dans la grille. Par convention, le pixel le plus en haut à gauche a la coordonnée (0,0) et le pixel le plus en bas à droite a la coordonnée (largeur-1,hauteur-1).

Chaque pixel est une cellule qui a une couleur spécifiée par ses composantes rouge, vert et bleu, qui sont des nombres entiers entre 0 et 255. Les trois composantes sont combinées pour déterminer la couleur finale du pixel. Plus une composante est élevée (jusqu'au maximum de 255), plus cette couleur sera présente dans la couleur finale du pixel. En JavaScript on peut donc se servir d'un enregistrement à trois champs pour représenter une couleur. Nous utiliserons les noms de champs `r`, `g` et `b` (pour red, green et blue). Voici comment on pourrait définir quelques couleurs :

```
var noir  = { r:   0, g:   0, b:   0 };
var blanc = { r: 255, g: 255, b: 255 };
var vert  = { r:   0, g: 255, b:   0 };
var jaune = { r: 255, g: 255, b:   0 };
```

Sous les ordres du pilote d'écran du système d'exploitation, le processeur graphique de l'ordinateur (le GPU), stocke dans chaque pixel les composantes r, g et b nécessaires pour afficher l'image voulue. Pour faire un affichage à l'écran, un programme doit envoyer des requêtes appropriées au système d'exploitation, dont le pilote d'écran traduira ces requêtes en commandes pour le GPU.

Dans l'environnement codeBoot l'écran graphique est simulé par une fenêtre qui apparait à la droite de la console. Chaque pixel de cet écran simulé peut être modifié pour y afficher une couleur précise. Quelques procédures et fonctions prédéfinies de codeBoot qui contrôlent cet écran graphique simulé seront utiles pour ce travail :

- `setScreenMode(`*largeur*`,` *hauteur*`)` : Conceptuellement, cette procédure établit la résolution de l'écran (comme une requête de changement de mode graphique envoyée au GPU). Dans codeBoot, cette opération est simulée en ajoutant à la droite de la console une grille de la largeur et de la hauteur demandées. Chaque carré de la grille représente un pixel de l'écran simulé. Pour faciliter la visualisation de l'écran simulé et le débogage, dans les faits un pixel simulé est un carré de quelques vrais pixels et il y a une bordure grise autour des pixels simulés afin de bien les distinguer. S'il y a un grand nombre de pixels la bordure grise disparait pour qu'on voit uniquement les pixels simulés.

- `getScreenWidth()` : Cette fonction retourne la largeur de l'écran simulé (le nombre de colonnes de pixels).

- `getScreenHeight()` : Cette fonction retourne la hauteur de l'écran simulé (le nombre de rangées de pixels).

- `setPixel(`*x*`,` *y*`,` *couleur*`)` : Change le contenu du pixel à la coordonnée (*x*,*y*) de l'écran simulé pour que sa couleur soit *couleur* (une structures avec trois champs `r`, `g` et `b` contenant des entiers entre 0 et 255).

- `getMouse()` : Retourne un enregistrement indiquant la position et l'état du bouton de souris. L'enregistrement contient 3 champs : `x`, `y` et `down`. Les champs `x` et `y` indiquent la coordonnée du pixel de l'écran simulé qui est pointé par la souris. Le champs `down` est un booléen qui sera `true` si et seulement si le bouton de souris est présentement appuyé.

- `exportScreen()` : Cette fonction retourne un texte qui encode le contenu de tous les pixels de l'écran créé par `setScreenMode`. Chaque pixel est représenté par un texte de 6 caractères donnant la valeur de chaque composante RGB en hexadécimal, préfixé par le caractère "`#`". Le caractère de fin-de-ligne "`\n`" sépare les rangées de pixels. Cette fonction est principalement utile pour écrire des tests unitaires. Voici un exemple d'utilisation à la console de codeBoot :

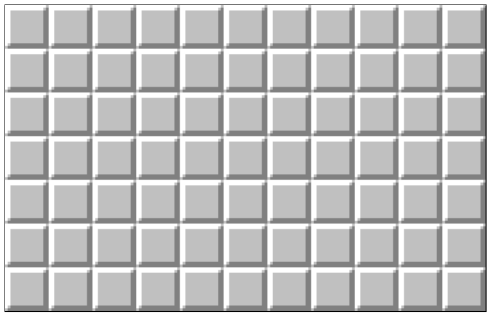
```
> setScreenMode(4,3)
> setPixel(2, 1, {r:255, g:128, b:5})
> exportScreen()
"#000000#000000#000000#000000\n#000000#000000#ff8005#000000\n#000000#000000#000000#000000"
> print(exportScreen())
#000000#000000#000000#000000
#000000#000000#ff8005#000000
#000000#000000#000000#000000
```

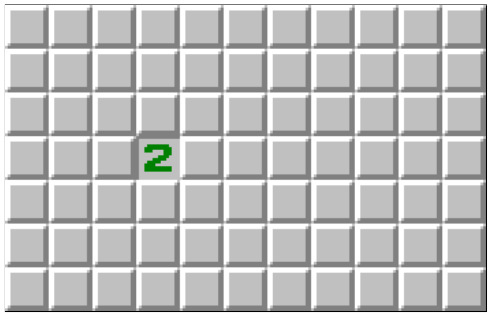
Dans le processus de développement, pour faciliter le débogage, il est conseillé d'utiliser une faible résolution d'écran simulé pour bien voir chaque pixel. C'est également utile pour faire des tests unitaires.

2 Jeu

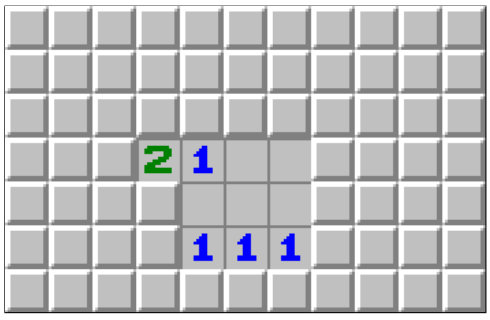
Le jeu de démineur est un jeu de réflexion qui consiste à découvrir par un processus d'élimination à quel endroit se trouvent des mines cachées sous des tuiles disposées en une grille. En cliquant sur une tuile on peut dévoiler ce qui est caché dessous. Si c'est une mine alors on perd le jeu. Sinon, la tuile dévoilée indique combien de tuiles directement voisines contiennent des mines. Les indices ainsi découverts permettent au joueur d'éviter les mines lors de ses prochains clics (jusqu'à un certain point). Pour gagner, le joueur doit dévoiler toutes les tuiles qui ne contiennent pas des mines.

Par exemple, voici une grille de tuiles de dimension 11x7 dans l'état initial du jeu (aucune tuile cliquée) à gauche, et à droite, après avoir cliqué sur une tuile à la coordonnée (3,3). Le jeu indique que 2 des 8 tuiles directement voisines de la tuile cliquée contiennent des mines.

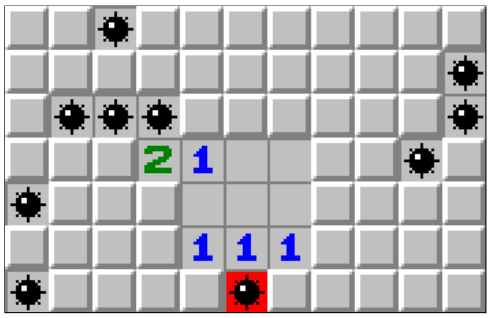




Si le joueur clique sur une tuile qui n'est pas une mine, et les tuiles directement voisines ne contiennent pas de mine, alors les tuiles directement voisines sont dévoilées en plus de la tuile cliquée. Remarquez qu'une tuile dévoilée qui n'a pas de mine voisine apparaît à l'écran comme une tuile vide plutôt que d'avoir l'image du chiffre 0. Voici par exemple ce que l'on verrait à l'écran après un deuxième clic sur une telle tuile à la coordonnée (5,4) :



Lorsque le jeu se termine, que ce soit par une victoire ou un échec du joueur, le jeu dévoile l'emplacement de toutes les mines. Si le joueur a perdu, la dernière tuile qu'il a cliquée (une mine) est affichée avec un fond rouge pour bien la distinguer des autres. Un message est aussi affiché avec `alert` pour indiquer la victoire ou l'échec du joueur. Voici ce qui pourrait être affiché lorsque le joueur clique sur une mine :



C'est le jeu qui place les mines aléatoirement. Pour être plus amusant, c'est au premier clic que la position des mines est établi. Le jeu s'assure que le joueur ne tombe pas sur une mine dès son premier clic.

3 Spécification

Vous devez concevoir et coder les procédures spécifiées ci-dessous en respectant le nom spécifié **exactement** pour qu'on puisse les tester plus facilement. Vous aurez sûrement à définir des fonctions et procédures auxiliaires (utilisez des noms appropriés de votre choix). Votre code doit être dans un fichier nommé "`demineur.js`".

Nous vous fournissons sur Studium un fichier "`images.js`" qui contient les définitions des images des 12 tuiles possibles (tuiles dévoilées avec 0 à 8 mines voisines, tuiles dévoilées d'une mine sans et avec fond rouge et la tuile non-dévoilée). Ce fichier contient la déclaration de 2 variables ayant comme valeur des tableaux : `colormap` et `images`. Le tableau `colormap` contient la définition des 10 couleurs qui sont utilisées dans les images de tuiles. Chaque élément du tableau `colormap` est une structure avec des champs `r`, `g` et `b` spécifiant la couleur. Le tableau `images` contient 12 éléments, chacun représentant une image. Une image est un tableau de 16 éléments, chaque élement représente une rangée de pixels de l'image et est un tableau contenant 16 entiers de 0 à 9 qui indiquent l'index de la couleur dans le tableau `colormap`. Chaque image contient donc 256 pixels (16x16 pixels). Les pixels sont disposés de haut en bas et de gauche à droite. Le tableau `images` est donc un tableau à 3 dimensions.

3.1 Procédure `afficherImage`(*x*, *y*, *colormap*, *image*)

Cette procédure affiche à l'écran à la coordonnée (*x*,*y*) l'image indiquée par le paramètre *image* qui utilise des couleurs définies par le paramètre *colormap*. Ce dernier doit être un tableau de tableaux d'entiers entre 0 et la longueur du tableau *colormap* dont les éléments sont des structures avec des champs `r`, `g` et `b` spécifiant des couleurs. La longueur du tableau *image* est la hauteur de l'image (en nombre de rangées de pixels) et la longueur des éléments du tableau *image* est la largeur de l'image (en nombre de colonnes de pixels). Les entiers contenus dans l'image sont donc des entiers qui sont l'index de la couleur du pixel dans le tableau `colormap`. L'image n'a pas nécessairement une taille 16x16 (en d'autres termes c'est une fonction générale pour afficher des images de n'importe quelle taille contenant n'importe quelles couleurs).

3.2 Fonction `attendreClic()`

Cette fonction attend que le bouton de souris soit relaché, puis attend que le bouton de souris soit appuyé. La fonction retourne un enregistrement contenant les champs `x` et `y` qui indiquent la coordonnée de la tuile sur laquelle le joueur a cliqué.

L'implantation de cette fonction doit se faire avec une boucle qui fait appel à la fonction `getMouse` et la fonction `pause` avec un paramètre égal à 0.01 . L'appel à la fonction `pause` permet de ne pas demander trop souvent l'état de la souris car ça pourrait gaspiller inutilement l'énergie (et chauffer le processeur).

3.3 Fonction `placerMines`(*largeur*, *hauteur*, *nbMines*, *x*, *y*)

Les paramètres *largeur* et *hauteur*, des entiers plus grands ou égal à 1, indiquent la taille de la grille en nombre de colonnes et rangées de tuiles respectivement. Le paramètre *nbMines* est un entier plus grand ou égal à 1 et plus petit que *largeur*hauteur*. Les paramètres x et y sont des entiers plus grands ou égal à 0 et plus petits que *largeur* et *hauteur* respectivement. La fonction `placerMines` retourne un tableau de *hauteur* tableaux de *largeur* booléens. Chaque élément de ce tableau à 2 dimensions indique s'il y a une mine sous la tuile à cette coordonnée. Tous les éléments sont `false` sauf pour *nbMines* éléments aléatoirement choisis qui sont `true`. L'élément qui correspond à la tuile à la coordonnée (x,y) est `false`. Faites attention à la performance de votre code lorsque *nbMines* est égal à *largeur*hauteur*−1.

3.4 Procédure `demineur`(*largeur*, *hauteur*, *nbMines*)

Cette procédure est la procédure principale du jeu. Les paramètres *largeur* et *hauteur* et *nbMines* sont des entiers avec les mêmes contraintes que la fonction `placerMines`.

La procédure `demineur` s'occupe du déroulement du jeu. Le jeu attend le premier clic, puis place aléatoirement les *nbMines* mines sous les tuiles, puis continue de dévoiler des tuiles en réaction aux prochains clics jusqu'à la fin du jeu.

Votre programme ne doit pas faire un appel à la procédure `demineur` car c'est le correcteur qui le fera. Pour votre phase de test, il est suggéré d'utiliser une petite grille, par exemple 5x3.

3.5 Procédure `testDemineur()`

Cette procédure effectue les tests unitaires de la procédure `afficherImage` et la fonction `placerMines`. Pour tester la procédure `afficherImage` il faudra utiliser la fonction `exportScreen` afin de pouvoir tester avec `assert` que le résultat est bon. Vous devez avoir de 5 à 10 tests par fonction et procédure.

Votre programme doit faire un appel à la procédure `testDemineur` pour lancer les tests. Aucun des tests ne doit faire une interaction avec l'utilisateur.

4 Évaluation

- Ce travail compte pour 15 points dans la note finale du cours. Vous devez faire le travail en équipes de 2 personnes. Indiquez vos noms clairement dans les commentaires au début de votre code.
- Vous devez remettre votre fichier "`demineur.js`" uniquement. La remise doit se faire au plus tard à 23h55 le lundi 5 novembre sur le site Studium du cours. Il y a une perte de 33% des points par jour (ou fraction de jour) de retard.
- Voici les critères d'évaluation du travail : l'exactitude (respect de la spécification), l'élégance et la lisibilité du code, la présence de commentaires explicatifs pour chaque fonction et dans le code, le choix des identificateurs, l'indentation du code respectant les standards de style, la décomposition

fonctionnelle, la performance (nombre de pas pour faire le traitement demandé) et l'utilisation d'un français sans fautes.

- Nous vous rappelons que toute forme de plagiat ne sera pas tolérée et entrainera une note de zéro pour le travail et possiblement d'autres sanctions. Tout le code dans ce travail (sauf celui fourni sur Studium) doit être conçu et codé par vous (ne copiez pas du code provenant du Web ou d'une autre équipe).