Université de Strasbourg
Programmation Web 1
Licence 1 - UFR Mathématique - Informatique
Automne 2019-20

Projet – Mise en œuvre d'un jeu de démineur

Objectifs du projet

- Mobiliser les connaissances acquises en cours et TP pour mettre en œuvre une application web basique ;
- Utiliser la documentation pour trouver les informations pertinentes pour répondre à un problème donné ;
- Présenter son travail de manière analytique dans un court rapport.

Instructions

- Lisez le sujet en entier (plusieurs fois) pour bien identifier les différents points sur lesquels vous devez travailler ;
- Ne faites que ce qui est demandé dans le sujet. Le hors sujet ne rapporte pas de point, même si vous montrez que vous êtes capable d'envoyer une fusée sur Mars (ça n'est pas ce qui est évalué dans cette UE) ;
- Lorsque c'est demandé, les choix de mises en œuvre doivent être justifiées : il faut convaincre votre correcteur que vous avez compris ce que vous avez développé ;
- Certains développements vous demanderont de chercher des informations dans la documentation. Vous devez montrer que vous savez trouver les informations pertinentes pour résoudre un problème donné.

Votre travail (codes sources et rapport) est à remettre au plus tard le 5 janvier 2020 à 20h sur l'espace Moodle prévu à cet effet. Aucun retard ne sera toléré.

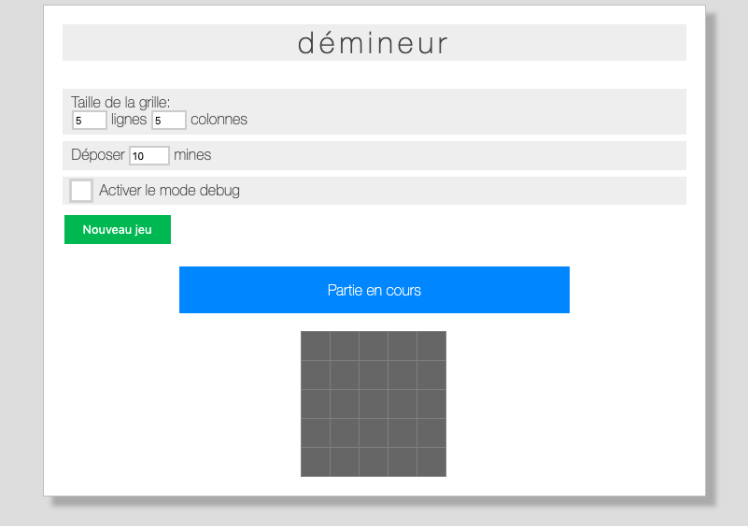

Figure 1 – Exemple d'application web "jeu de démineur"

1 Jeu de démineur, quésaco ?

Wikipédia nous informe que [1]

> Le démineur est un jeu de réflexion dont le but est de localiser des mines cachées dans un champ virtuel avec pour seule indication le nombre de mines dans les zones adjacentes.

Votre objectif est de mettre en œuvre un jeu de démineur avec HTML/CSS/Javascript, tel qu'illustré sur la figure 1. Vous devez développer une interface web (HTML/CSS) à laquelle des comportements (Javascript) sont attachés pour afficher une grille contenant des cellules (contenant des mines ou non) sur lesquelles l'utilisateur/ice va cliquer. Si il/elle tombe sur une mine, la partie est perdue. Si il/elle parvient à dévoiler toutes les cellules ne contenant pas de mines, l'utilisateur/ice gagne la partie.

Pour aider l'utilisateur/ice, chaque cellule ne contenant pas de mine contient le nombre de cellules voisines contenant une mine. Le voisinage utilisé pour cela est un voisinage de Moore d'ordre 1 [2] tel qu'illustré sur la figure 2 (haut, bas, gauche, droite ainsi que diagonale haut/gauche, diagonale haut/droit, diagonale bas/gauche et diagonale bas/droite –soit 8 voisins)

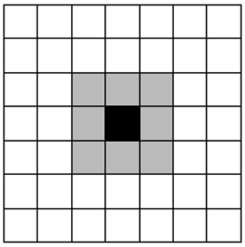

FIGURE 2 – Voisinage de Moore : la cellule examinée est la cellule noire qui a pour cellules voisines les cellules grises.

2 Structure HTML (≈4 points)

Un exemple d'interface (donné à titre indicatif) est illustré sur la figure 1.

Votre structure HTML doit contenir les éléments suivants :

1. Un titre
2. Des champs pour renseigner
   (a) les dimensions de la grille de jeu : vous devez utiliser des éléments `input` de type "number" et spécifier une taille minimale (resp. maximale) de 2 (resp. 20) dans ce champs en utilisant les attributs adéquats ;
   (b) le nombre de mines dans la grille. Là aussi, il faut utiliser un champs de type "number". Il faut au moins une mine déposée, mais pas de contraintes (au niveau du formulaire) sur la borne maximale ;
   (c) une case à cocher pour activer le mode "débugage" (ce mode est détaillé dans la partie 4.5) ;
   (d) un bouton démarrer une partie avec les paramètres de la grille.

1. `https://fr.wikipedia.org/wiki/D%C3%A9mineur_(genre_de_jeu_vid%C3%A9o)`
2. `https://fr.wikipedia.org/wiki/Voisinage_de_Moore`

3. Un espace pour afficher le statut du jeu (partie en cours, perdue, gagnée, ...).
4. Un espace pour la grille de jeu. La grille de jeu est composée de lignes, elles mêmes composées de cellules. Votre structure HTML doit simplement comporter un conteneur "grille" (les lignes et cellules seront ajoutées dynamiquement par modification du DOM avec Javascript).

Il est attendu que vous présentiez et justifiez (en une demi-page) votre structure HTML ("parce que c'est demandé dans le sujet" n'est pas une justification valide. Nous avons assez discuté en cours de l'importance de la structuration HTML pour que vous soyez capable d'expliquer vos choix de mise en œuvre).

3 Formatage et agencement avec CSS (≈4 points)

3.1 Formatage de la page

Vous devez créer des styles (associés à des classes/identifiants spécifiés dans votre structure HTML) afin de satisfaire les demandes suivantes :

1. Le document doit être écrit avec une police sans sérif ;
2. Le titre de la page doit être centré dans le corps du document ;
3. Le titre de la page et les éléments du formulaires doivent avoir pour couleur de fond `#eeeeee` ;
4. Votre bouton doit avoir une couleur de fond verte, un texte écrit en blanc, un *padding* en haut et en bas de 5px, à droite et à gauche de 10px (le tout défini par une seule définition de la propriété `padding`) ;
5. Le statut du jeu doit avoir pour taille 50% du *viewport* (100% en version "mobile" – *cf.* partie 3.2) et avoir une couleur de fond (à votre convenance) ;
6. Les cellules de la grille doivent avoir la couleur de fond `#666`, une bordure grise de 1 pixel continue. Le texte des cellules doit être centré (verticalement *et* horizontalement). Vous devez gérer le fait que les bordures vont se superposer [3] Expliquez la solution mise en œuvre.

3.2 Agencement avec CSS Flexbox

Toute votre page doit être agencée avec des Flexbox. Pour cette partie, vous êtes libres d'agencer vos éléments comme bon vous semble, mais vous devez (1) mettre en œuvre une *media query* et (2) expliquer clairement vos choix d'agencement (agencement désiré et propriétés utilisées pour la mise en œuvre).

N'appliquez pas votre *media query* sur la grille de jeu en elle-même... Tant pis si l'utilisateur/ice doit scroller horizontalement (une gestion plus fine induirait une gestion plus complexe des voisinages et ça n'est pas du tout à l'ordre du jour de ce projet).

4 Comportements de l'application avec Javascript (≈12 points)

4.1 Quelques structures de données et variables

Au niveau global, vous utiliserez une grille "algorithmique" (permettant de gérer le déroulement du jeu) avec la variable `var grille = [];` qui vous permettra de gérer votre grille de jeu dans le code

3. Conseil : cherchez "css border collapse" sur votre moteur de recherche préféré. N'utilisez pas `display: table` pour gérer le chevauchement des bordures ! Un simple travail sur les marges suffit.

Javascript. Cette grille sera bien en deux dimensions, même si sa déclaration laisse penser le contraire : lors de la création de la grille de jeu (section 4.2), vous ajouterez des lignes (comportant plusieurs cellules) à la grille "algorithmique".

Vous devez aussi utiliser quelques booléens pour gérer le déroulement de la partie : `var jeu_fini = false;`, `var debug_actif = false`. Enfin, cela sera aussi utile de conserver globalement le nombre de cellules déjà révélées par l'utilisateur/rice.

4.2 Nouvelle partie : création d'une grille

Les cellules de la grille Votre grille "algorithmique" est composée de cellules. Chaque cellule est caractérisée (a) par sa valeur –soit une mine, *i.e.* une valeur > 8–, soit le nombre de mines dans le voisinage de la cellule– et (b) par un booléen permettant d'indiquer si le joueur a révélé le contenu de la cellule. Pour faciliter la gestion du jeu, vous devez créer un objet `Cellule` (en utilisant un constructeur, tel que vu en cours) contenant les deux propriétés (a) et (b).

Algorithme pour la création d'une grille L'algorithme à mettre en œuvre pour créer une grille est le suivant :

Algorithme 1 : Création d'une grille

lignes ← entrée de formulaire
colonnes ← entrée de formulaire
nb_mines ← entrée de formulaire
si *nb_mines* > *dimensions de la grille* **alors retourner**-1

grille_html ← récupérer élément HTML "grille"
pour chaque $l \in$ *lignes* **faire**
  ligne_html ← créer un élément HTML "ligne"
  ligne_algorithmique ← `[] /* création d'un tableau */`
  pour chaque $c \in$ *colonnes* **faire**
    cellule_html ← créer un élément HTML "cellule"
    attacher l'élément *cellule_html* à l'élément *ligne_html* `/* modification du DOM */`
    ajouter une nouvelle instance de `Cellule` à *ligne_algorithmique* `/* méthode push() */`
  fin
  attacher l'élément *ligne_html* à l'élément *grille_html* `/* modification du DOM */`
  ajouter *ligne_algorithmique* à *grille_algorithmique* `/* méthode push() */`
fin

placer_mines(*nb_mines*, *lignes*, *colonnes*) `/* implémentée dans la suite */`
voisinage() `/* implémentée dans la suite */`

Démarrer une nouvelle partie Maintenant que vous avez tout ce qu'il faut pour créer une grille, il est temps de permettre à l'utilisateur de démarrer une nouvelle partie. Pour ce faire, les étapes à mettre en œuvre dans la fonction `function nouvelle_partie()` sont les suivantes :

1. Remettre le booléen `jeu_fini` à faux ;
2. Réinitialiser la variable globale `grille_algorithmique` ;
3. Enlever, du DOM, tous les enfants de l'élément HTML "grille" ;

4. Remettre le booléen `debug_actif` à faux ;
5. Décocher la case à cocher pour activer le mode débugage ;
6. Appeler la fonction de création de la grille. Si la valeur -1 est renvoyée, il faut afficher un message d'erreur dans l'élément HTML "statut".

Cette fonction doit être appelée lorsque le bouton "Nouveau jeu" est cliqué.

4.3 Placer des mines

Pour l'instant, on travaille dans la grille "algorithmique". On propose pour cette fonction le prototype suivant : `function placer_mines(nb_mines, nb_lignes, nb_colonnes)`. Les n mines sont placées aléatoirement en utilisant la fonction suivante [4] :

```
/* Returns a random integer between min (included) and max (excluded) */
function getRandomInt(min, max) {
  min = Math.ceil(min);
  max = Math.floor(max);
  return Math.floor(Math.random() * (max - min)) + min;
}
```

Les cellules aléatoirement sélectionnées pour contenir une mine devront contenir une valeur identifiable (un entier strictement supérieur à 8 –le nombre maximal de voisins contenant une mine qu'une cellule donnée peut avoir).

4.4 Calculer les voisinages des cellules sans mines

On travaille toujours sur la grille "algorithmique". Le prototype proposé pour cette fonction est `function voisinage()`. Il faut calculer, pour chaque cellule de la grille qui ne contient pas de mine, le nombre de mines dans les cellules voisines (selon un voisinage de Moore tel qu'illustré plus haut) en utilisant l'algorithme 2. Attention ! Pensez au fait que les cellules sur les bords de la grille n'ont pas exactement 8 cellules voisines...

Algorithme 2 : Voisinage d'une cellule

```
pour chaque cellule de coordonnées (l, c) de la grille algorithmique faire
    si la cellule (l, c) ne contient pas de mine alors
        pour chaque cellule de coordonnées (i, j) voisine de la cellule (l, c) faire
            si la cellule (i, j) contient une mine alors
                incrémenter la valeur contenue dans la cellule (l, c)
            fin
        fin
    fin
fin
```

4.5 Avant de continuer : le mode *débug*

En mode débug, toutes les informations de la grille doivent être affichées (emplacement des mines ou nombre de mines dans les cellules voisines), comme illustré sur la figure 3.

4. Extraite de `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/random`

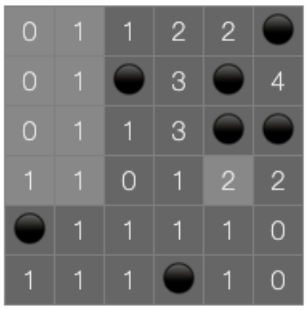

FIGURE 3 – Affichage HTML en mode débug : les mines ou le nombre de cellules voisines contenant des mines est affiché.

Une fonction pour vous aider Vous allez avoir besoin de faire le lien entre votre grille algorithmique et votre grille HTML : lorsque vous allez parcourir la grille algorithmique pour, par exemple, accéder aux valeurs contenues dans les cellules, il faudra afficher cette information dans la cellule correspondante dans la grille HTML.

Expliquer les différentes opérations réalisées dans la fonction suivante, et dîtes à quoi cette fonction vous servira :

```
function la_fonction(ligne, colonne){
  var ligne_html = $(".ligne").eq(ligne).children();
  return ligne_html.slice((colonne+1)-1, (colonne+1));
}
```

Activer / désactiver le mode débug Lorsque vous cochez la case pour l'activation du mode débug dans votre formulaire, il faut parcourir l'ensemble de la grille algorithmique. Pour chacune des cellules de cette grille, vous devez accéder à la cellule html correspondante dans le DOM et, selon l'état de la case à cochée (cochée ou non), afficher (ou masquer) les informations contenues dans les cellules de la grille algorithmique dans la grille HTML.

4.6 C'est l'heure de jouer ! Révéler des cases / version 1

Rappels Pour jouer au démineur, vous allez cliquer sur les cellules de la grille HTML. Les cellules étant insérées dynamiquement dans le DOM, rappelez comment vous allez gérer la détection de l'événement "clic de souris" sur ces cellules.

Révéler le contenu d'une cellule cliquée Votre objectif est maintenant de mettre en œuvre la fonction `function reveler_cellule(cellule_html /*cellule cliquée*/)`. Gardez en tête que les cellules sont des éléments enfants des lignes qui sont elles mêmes des éléments enfants de la grille : comment allez-vous récupérer les coordonnées (la ligne et la colonne) de la cellule cliquée ?

Une fois récupéré les coordonnées de l'élément HTML cellule cliqué, vous allez pouvoir utiliser les informations contenues dans la grille algorithmique :

1. Si la valeur de la cellule cliquée dans la grille algorithmique est strictement supérieur à 8 (*cf.* section 4.3), alors on a cliqué sur une cellule contenant une mine : le jeu est terminé et la fonction retourne 1.

2. Sinon, dans la grille algorithmique, il faut passer à vrai le booléen indiquant si la cellule a été cliquée. Pour la cellule HTML cliquée elle-même, il faut afficher la valeur de la cellule (contenue dans la grille algorithmique) et passer la couleur de la cellule à `#888` (pour indiquer visuellement que la cellule a été cliquée) et retourner 0.

Boucle principale du jeu C'est simple : le jeu continue (*i.e.* on peut continuer à révéler des cellules) tant que (1) on n'a pas cliqué sur une mine –si `reveler_cellule()` renvoie 1, alors le jeu est terminé– ou (2) il reste des cellules ne contenant pas de mines à révéler.

4.7 C'est l'heure de jouer ! Révéler des cases / version 2

Une version plus avancée de la fonction `reveler_cellule()` consiste à révéler toute une zone de cellules autour de la cellule cliquée : si celle-ci n'a aucune cellule voisine contenant des mines, alors on peut afficher toutes ces cellules voisines et répéter l'opération jusqu'à ce qu'une des cellules voisines de la cellule examinée contienne une mine. La zone révélée est alors délimitée par des cellules ayant des voisines contenant une mine (*cf.* figure 1).

Créer la fonction `function reveler_cellule(cellule_html)` permettant de révéler *récursivement* toutes les cellules voisines de `cellule_html` ne contenant pas de mines : il faut appliquer l'appel récursif sur chacune des 8 cellules voisines de la cellule de coordonnées (l, c) passée en paramètre. On arrête les appels récursifs si la cellule courante a une valeur (dans la grille algorithmique) supérieure à 0 –*i.e.* si elle contient une ou des cellules voisines contenant une mine.