Licence Sciences technologies santé mention informatique parcours Informatique générale

NFP 119

Programmation Fonctionnelle : des concepts aux applications web

Projet 2024

Gregoire BURNET

Introduction

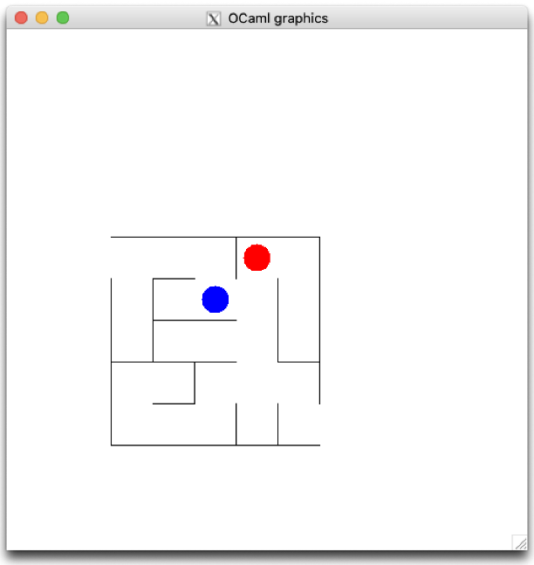
Le rendu de ce projet est découpé en plusieurs partie, indépendante l'une de l'autre dans une certaine mesure l'ensemble de ce que vous produirez dans le cadre de ce projet sera à rendre dans devoir sur Moodle dans une archive au format zip ou rar ou 7z au choix.

Vous devez faire ce travail seul sans chercher de solution sur internet. Le compte rendu a pour objectif entre autres d'évaluer votre compréhension du code que vous avez rendu. En particulier il vous sera demande d'écrire de nouvelles fonctions pour le projet.

PARTIE I. Projet sous Ocaml

I. Description générale

Le but du projet est de programmer un petit jeu qui se présente sous la forme suivante



Le joueur humain déplace le disque bleu (que l'on appelera par la suite pacman) `a l'aide du clavier. Initialement pacman est dans l'angle en haut `a gauche et il doit se rendre dans l'angle en bas a droite du labyrinthe sans traverser de mur. Pour mettre un peu de piment, le disque rouge (que l'on appelera par la suite le fantome) pourchasse pacman et tente de lui barrer la route. Initialement, le fantome est dans l'angle en haut a droite, et `a intervalle de temps regulier (par exemple, toutes les deux secondes) il se deplace d'une case de facon a se rapprocher de pacman.

Le projet est decoupe en trois parties.

1. Generation aleatoire du labyrinthe et affichage. Il s'agit d'une partie de difficulte moyenne mais tres guidee, qui compte pour environ 50% de la note finale.

2. Gestion du clavier et du pacman. Il s'agit d'une partie relativement facile qui compte pour environ 25% de la note finale.
3. Gestion du fantome. Il s'agit d'une partie un peu plus difficile mais tres guidee qui compte pour environ 25% de la note finale.

II. La génération du labyrinthe

On souhaite pouvoir generer un labyrinthe de dimension arbitraire - ultimement cela sera le joueur qui fixera la dimension du labyrinthe au debut de la partie.

On souhaite que le labyrinthe genere soit "difficile" au sens où il existe

un seul chemin entre le depart et l'arrivee, et plus generalement entre deux cases quelconques du labyrinthe. Suivant la terminologie de wikipedia, un tel labyrinthe sera dit *parfait*. Pour garantir que le labyrinthe genere est parfait, on applique l'algorithme suivant: on part de la grille pleine avec des murs partout, puis on retire des murs un par un en les tirant au hasard. Cependant, un mur n'est retire que si cela permet de creer un chemin entre les deux cases voisines qu'il separe; si les deux cases sont deja reliees par un chemin detourne, on ne retire pas le mur et on tire au hasard un nouveau mur, et ce jusqu'à en trouver un convenable. Au total, on retire un nombre de murs egal au nombre de cases moins 1. Une animation qui explique l'algorithme est visible sur la page wikipedia consacree à la generation de labyrinthes.

La partie subtile de l'algorithme est le test pour savoir si les deux cases sont deja reliees par un chemin. Pour cela, on commence par numeroter les cases: les cases de la premiere ligne ont les numeros de 0 à $l-1$, où l designe la largeur du labyrinthe ($l = 5$ sur l'exemple); les cases de la ligne suivante ont les numeros l à $2l-1$, etc. Ainsi la case du pacman dans la figure ci-dessus sera la numero 7.

On utilise ensuite une structure de donnees uf dite *union-find* car elle permet deux operations:

- find uf n : cette operation renvoie le numero d'une case qui est actuellement reliee par un chemin a la case n. La particularite de find est que toutes les cases d'une meme composante connexe prennent la meme valeur pour find. Autrement dit, pour tester si deux cases n et m sont reliees par un chemin, il suffit de tester si find uf n = find uf m.
- union uf n m: cette operation modifie la structure de donnees uf (et donc le calcul de find) de sorte que les cases n et m soient maintenant considerees comme faisant partie de la meme composante connexe. Cette operation est utilisee quand on retire le mur qui separe les cases n et m pour fusionner les deux composantes connexes en une seule.

La structure de donnees uf se presente comme une foret avec un arbre par composante connexe (en interne, on code cette foret à l'aide d'un tableau). Le resultat de find uf n est la racine de l'arbre dans lequel se trouve n. Pour gagner en efficacite, on cherche à avoir des arbres aussi peu profonds que possible, ce qui fait appel à deux heuristiques: la compression de chemin et l'equilibrage au moment de

la fusion. Tous les details sur la structure de donnees union-find se trouvent sur la page wikipedia.

Pour representer le labyrinthe, on va remplir un tableau de booleens `a trois entrees: mur_present.(d).(x).(y) sera vrai si le mur de direction d (0 pour vertical, 1 pour horizontal), ligne x et colonne y est present. Par exemple, sur le labyrinthe represente ci-dessus, on a mur_present.(0).(2).(0) = true car le mur vertical de la colonne 2 et la ligne 0 est present - il s'agit du mur `a gauche du fantome, on compte les colonnes `a partir de la premiere rangee de murs interne, en sautant la rangee de murs externes.

Pour tirer un mur au hasard, on pourra tirer un entier au hasard entre 0 et #murs-1, o`u #murs est le nombre total de mur, soit $(l-1)h + l(h-1)$.

```
let mur_au_hasard l h = (* renvoie un triplet (d, x, y) *)
  let n = Random.int ((l-1) * h + l * (h-1)) in if n < (l-1) * h
  then (0, n mod (l-1), n / (l-1)) else let n2 = n
  - (l-1) * h in
     (1, n2 mod l, n2 / l)
```

Parlons maintenant de l'affichage. Vous allez utiliser la librairie Graphics que vous avez deja vue a certains TP, et dont la documentation se trouve en ligne. Il faudra compiler votre programme avec quelque chose comme ocamlc graphics.cma projet.ml. Pensez a faire un appel a read_key() en fin de programme pour que l'affichage soit visible tant que vous n'avez pas press´e un bouton. Vous aurez essentiellement a utiliser les fonctions open_graphics, moveto, et lineto pour le trace du labyrinthe. Il faudra d'abord tracer le pour- tour du labyrinthe, puis parcourir le tableau mur_present qui code le labyrinthe, et pour chaque mur tel que mur_present.(d).(x).(y) est vrai, il faudra af- ficher le mur correspondant. Vous aurez a manipuler les parametres suivants, qui devront etre instancies par des constantes au dernier moment (idealement, il faudrait les demander a l'utilisateur avant de commencer):

- upleftx et uplefty : les coordonnees de l'angle en haut a gauche du labyrinthe
- taille_case : la largeur (et la hauteur) d'une case en pixel
- l et h, la largeur et la hauteur du labyrinthe en nombre de cases

Les questions suivantes visent a vous guider. Vous n'etes pas oblige d'y repondre si vous voulez faire autrement, l'essentiel est d'arriver a generer le labyrinthe et a l'afficher.

https://fr.wikipedia.org/wiki/Union-find

https://ocaml.github.io/graphics/graphics/Graphics/index.html

https://fr.wikipedia.org/wiki/Modelisation_mathematique_de_labyrinthe

1. Definissez un module UF qui implemente une structure de donnees union- find de type abstrait t et offre trois fonctions:
   - create n renvoie une nouvelle structure de donnees uf de type t. Le parametre n designe les entiers de 0 `a $n-1$ qui vont etre fusionnes. Initialement, find uf i renvoie i pour chaque i dans cette structure de donnees (il n'y a encore eu aucune fusion).
   - find uf n qui renvoie le representant de la classe de n dans la structure d'union-find uf
   - union uf n m qui modifie uf pour faire fusionner les classes de n et m.
2. Definissez la fonction cases_adjacentes l h (d,x,y) qui au mur (d,x,y) associe le couple (i,j) des numeros des deux cases separees par ce mur. l et h sont la largeur et la hauteur du labyrinthe en nombre de cases. Par exemple, sur un labyrinthe 5×5 comme dans l'exemple, cases_adjacentes 5 5 (0,0,2) renvoie (10,11), tandis que cases_adjacentes 5 5 (1,1,0) renvoie (1,6).
3. Definissez la fonction generate_lab l h qui renvoie un tableau mur_present a trois entrees codant les murs du labyrinthe. Pour vous aider un peu, on vous donne cette fonction en pseudo-code allouer le tableau mur_present en mettant toutes les cases `a true allouer uf avec UF.create (l*h)

```
pour i allant de 1 à l*h-1 faire
   soit m un mur au hasard
   soit i et j les cases adjacentes separees
   par m si UF.find uf i = UF.find uf j
   alors reprendre la boucle au début sans incrémenter
   i sinon faire
     UF.union uf i j
     avec          m=(d,x,y)          faire
       mur_present.(d).(x).(y) <- false
   fin sinon fin pour
renvoyer mur_present
```

4. Definissez la fonction trace_pourtour upleftx uplefty taille_case l h qui dessine le pourtour du labyrinthe.
5. Definissez la fonction trace_mur upleftx uplefty taille_case (d,x,y) qui dessine le mur (d,x,y).
6. Definissez la fonction trace_lab upleftx uplefty taille_case l h mur_present qui dessine le labyrinthe cod´e par le tableau de booleens mur_present.

Si votre code est correct, vous devriez voir s'afficher un labyrinthe parfait, sans partie fermee et sans "salles" puisqu'il y a toujours exactement un seul chemin entre deux cases.

III. Le pacman

Pour gerer le pacman, il sera commode d'avoir une variable globale case_pacman qui contient le numero de la case du pacman, initialement en haut a gauche sur la case 0.

```
let case_pacman = ref 0
```

Cette variable globale sera modifiee a chaque pression d'un bouton. La bibliotheque Graphics offre la fonction read_key. Vous ne pourrez malheureuse- ment pas recuperer les touches de fleches, la fonction ne renvoyant que les car- acteres imprimables. Le plus simple sera donc de deplacer le pacman avec des touches correspondant a des lettres. Sur mon clavier americain, j'ai pris 'a' pour gauche, 'w' pour haut, 's' pour droit, et 'z' pour bas.

La fonction read_key sera appellee dans une boucle while infinie. En dehors de l'appel `a la fonction read_key, les actions effectuees dans cette boucle sont:

- analyse de la touche pressee et verification que le mouvement est autorise (pas de mur)
- mise `a jour de case_pacman
- mise `a jour de l'affichage
- test si la case finale est atteinte, et sortie du jeu en affichant "GAGNE" si c'est le cas

Pour mettre a jour l'affichage, il y a deux strategies possibles: soit vous effacez toute la fenetre avec clear_graph et vous redessinez tout, soit vous effacez seulement la derni`ere position du pacman (en dessinant par exemple un disque blanc dessus) et vous dessinez pacman a sa nouvelle position.

Bonus: si vous voulez, vous pouvez emettre un son quand pacman se cogne a un mur. Vous pouvez aussi rajouter une touche pour terminer la partie sans fermer la fenetre (par exemple la touche 'q' pour 'quitter').

IV. Le fantome

Comme pour le pacman, il sera commode pour le fantome d'avoir une variable globale qui contient le num´ero de la case sur laquelle il se trouve. Initialement, le fantome se trouve sur la case en haut a droite de numero $l - 1$. En supposant que la largeur du labyrinthe est elle aussi stockee dans une variable globale l, on aura

```
let case_fantome = ref (l-1)  (* ou !l -1 *)
```

Le coeur du code du fantome est une boucle while infinie qui execute les actions suivantes:

- s'endormir un laps de temps (par exemple 2 secondes)
- calculer la prochaine case o`u se rendre et mettre `a jour case_fantome
- mettre `a jour l'affichage

- tester si la case du pacman est atteinte et si oui quitter la partie en affichant ”PERDU”

Le test pour savoir si la partie est perdue doit d'ailleurs etre fait aussi dans la boucle infinie du pacman car cela peut etre un mouvement de pacman vers le fantome qui met fin a la partie. C'est l'occasion de refactoriser le code et d'ecrire des fonctions pour les parties communes aux deux boucles: affichage, test si perdu, et sortie du programme.

Vous vous demandez peut-etre comment il peut y avoir deux boucles infinies dans le programme... bonne question! La boucle infinie du fantome doit en effet etre executee en tache de fond. Pour ce faire, il faut la faire executer par un thread different du thread principal, le thread principal etant celui qui execute la boucle infinie du pacman.

on pourrait aussi mettre la boucle du pacman dans un thread secondaire, mais cela n'a aucun interet, et il faut veiller a ce que le thread principal ne termine pas pour ne pas risquer de terminer tous les autres threads.

Pour illustrer l'utilisation des threads et d'un minuteur, voici un programme assez analogue a ce qui nous interesse. Ce progamme attend la saisie d'un texte avec read_line et l'affiche (l'analogue du pacman) tandis qu'en tache de fond un autre thread affiche ”hello” toutes les deux secondes (l'analogue du fantome).

```
let rec affiche_hello () = Unix.sleep 2;
  print_string "hello" ; print_newline(); affiche_hello()

let _ = Thread.create affiche_hello () let () =

(* thread principal *)
  let s = read_line () in
  print_string s
```

Pour compiler ce programme, taper

ocamlc -thread unix.cma threads.cma hello.ml

où hello.ml est le nom du fichier qui contient ce code. De meme, pour compiler votre projet, il vous faudra reprendre ces options ainsi que la librairie graphics:

ocamlc -thread graphics.cma unix.cma threads.cma projet.ml.

Dans un premier temps, simplifiez le calcul de la prochaine case pour pouvoir tester votre programme avec le thread; par exemple vous pouvez faire un fan- tome qui traverse les murs et qui se dirige en ”ligne droite” vers le pacman.

Pour etre tout a fait juste, le programme comportera un petit bug que l'on va s'autoriser: si la mise a jour du pacman et du fantome se produisent en meme temps, on peut observer des comportements bizarres, par exemple le fantome peut etre affiche a deux positions differentes simultanement. Pour resoudre ce probleme il faudrait utiliser un verrou et vous expliquer comment cela fonctionne. Ce n'est pas tres

complique mais c'est hors programme; on va simplifier et ne pas se soucier de ce bug qui a de toute facon tres peu de chance de se produire.

Passons maintenant au calcul de la prochaine case où doit se rendre le fan- tome. A tout moment le pacman et le fantome sont relies par un chemin qui est unique. La case sur laquelle le fantome doit se rendre est la premiere case de ce chemin. Pour la calculer, on va supposer que l'on connait pour chaque case c la liste voisines(c) des cases voisines accessibles en un pas; on pourra remplir un tableau voisines a l'interieur de la fonction generate_lab et le renvoyer en meme temps que le tableau mur_present.

La fonction principale pour la recherche de chemin est la fonction est_relie src dst evite voisines qui renvoie vrai s'il existe un chemin allant de la case src a la case dst sans passer par la case evite en premier. On a besoin de rajouter cette case a eviter pour ne pas creer une boucle infinie dans la recherche. La fonction est_relie src dst evite voisines procede comme suit:

```
si src=dst alors renvoyer true fin si.
pour toute case c dans voisines.(src), c<>evite, faire
    si est_relie c dst src voisines alors renvoyer true fin si fin pour
renvoyer false
```

PARTIE II. Projet en JavaScript

Objectif

Le projet consiste en un petit jeu dans lequel le but est de cliquer en un minimum de temps sur des cibles disposées aléatoirement dans une zone délimitée.

Présentation

Comme l'illustre la figure ci-dessous on identifie différents éléments dans la page HTML. D'abord au centre se trouve la zone où apparaissent les cibles qu'il faudra faire disparaitre par un clic, cet élément a pour `id` *terrain*. Ensuite on trouve en haut de la page un premier bouton intitulé Une cible. Un clic sur ce bouton crée une unique cible qui apparaît à une position aléatoire sur le terrain. Enfin, dans la partie inférieure se trouve la zone de contrôle du jeu. On y trouve différents éléments :

- un champ de saisie numérique pour indiquer le nombre de cibles à créer,
- un bouton Démarrer qui provoque la création des cibles et le début du jeu,
- une zone d'affichage du nombre de cibles qu'il reste à cliquer,
- une zone d'affichage du chronomètre, le temps est affiché en minutes, secondes et dixièmes de secondes.

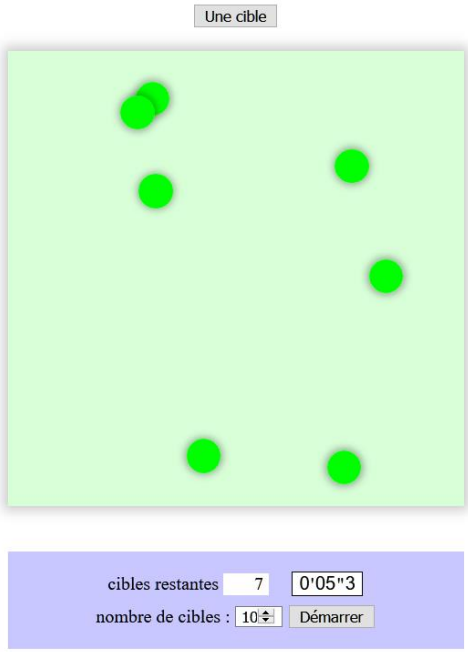


Figure 1 : Aperçu du projet

Un clic sur le bouton Démarrer permet de commencer le jeu, ce qui a pour conséquence de :

- supprimer toutes les cibles qui pourraient rester dans le terrain,
- stopper et remettre à zéro le chronomètre,

- créer le nombre de cibles correspondant à la valeur indiquée par le joueur dans la zone dédiée. Les cibles doivent être positionnées aléatoirement dans le terrain.
- mettre à jour la zone affichant le nombre de cibles qu'il reste à détruire.

Comme montré dans la vidéo, quand toutes les cibles ont été cliquées, le chronomètre s'arrête et le temps réalisé est annoncé (pour une meilleure esthétique vous pouvez créer votre propre zone d'annonce plutôt que le résultat pas très joli du *window.alert* de la vidéo).

Les cibles

Les cibles créées correspondent à des éléments *div* de classe CSS *target*. En consultant le code CSS vous pouvez vérifier que pour ces éléments la propriété *position* a pour valeur *absolute*. On peut donc positionner précisément ces éléments dans la zone de terrain en donnant des valeurs aux propriétés de style *top* et *left*. Lorsqu'il est créé un élément cible est ajouté comme nouvel enfant de l'élément *#terrain* et positionnée aléatoirement dans celui-ci. Au moment de son ajout une cible se voit ajouter la classe CSS *on* (en plus de la classe *target*).

Les cibles doivent être cliquables. Lorsque l'on clique sur une cible, le nombre de cibles restantes qui est affiché doit être mis à jour. Le clic sur une cible donnée ne doit être actif qu'une seule fois. Au moment du clic, la classe CSS *hit* est ajoutée à un élément cible pour obtenir un changement visuel. Enfin un élément cible cliqué doit disparaître et être retiré des enfants de l'élément *#terrain* 1 seconde après avoir été cliqué. Vous utiliserez pour cela la fonction *window.setTimeout* dont vous aurez étudié la documentation sur le MDN.

Chronomètre

La réalisation du jeu nécessite la mise en place d'un chonomètre. Celui sera géré à l'aide de la fonction *window.setInterval* qui crée un timer et qui a déjà été utilisée pour l'exercice sur les diaporamas. Le timer utilisé sera mémorisé dans la variable globale *chronoTimer* définie dans le fichier scripts/project2018.js présenté ci-dessous. Une fois démarré ce timer aura pour action d'incrémenter tous les *100ms* la variable globale *time*. C'est grâce à la valeur de cette variable que l'on pourra mesurer le temps écoulé.

Le timer a également pour rôle la gestion de l'affichage du temps écoulé dans les différents éléments de la zone d'id `chrono`.

Le chronomètre est stoppé quand le jeu se termine (toutes les cibles ont été cliquées) et est réinitialisé à chaque clic sur le bouton Démarrer.

Travail à réaliser

Le travail que vous devez réaliser consiste à écrire le code javascript permettant de mettre en œuvre les comportements décrits ci-dessus.

Vous devez utiliser les fichiers contenus dans l'archive fichiers-mini-projet-js202x.zip fournie. Cette archive contient :

- le fichier mini-projet-js-201x.html que vous renommerez en *votreNom*-projet.html mais dont **vous ne modifierez pas le contenu**,
- le fichier style/style-project201x.css, vous n'avez a priori pas à le modifier non plus,
- le fichier scripts/project201x.js que vous compléterez avec le code javascript que vous écrirez. Il contient déjà les définitions de variables globales que vous utiliserez : les largeur et hauteur des cibles (conformément avec la feuille CSS) ainsi que les variables utiles pour la gestion du chronomètre et évoquées ci-dessus.
  Toutes les fonctions que vous écrirez dans ce fichier devront être documentées et leur code commenté quand nécessaire.

4. Etudiez le code html du document mini-projet-js-201x.html afin de bien en comprendre la structure et de repérer les différents éléments qui constituent la page, et notamment leurs `id`.
5. Mettez en place la gestion du bouton Une cible. Il vous faut donc gérer :
   - la création d'un élément cible, son positionnement à des coordonnées aléatoires et son ajout comme enfant de l'élément *#terrain*,
   - l'effet du clic sur cet élément et sa disparition après 1 seconde.
6. Créez les fonctions nécessaires à la gestion du chronomètre et faites en sorte qu'un clic sur le bouton Démarrer démarre le chonomètre et son affichage dans la zone *#chrono*.
7. Ajoutez les fonctions permettant le déroulement du jeu. Vous devrez donc compléter l'action du clic sur le bouton Démarrer pour qu'il provoque la création du nombre de cibles voulues et également enrichir l'effet du clic sur les cibles pour diminuer le nombre de cibles restantes affiché et détecter la fin du jeu.
8. (options) Vous pouvez de manière **optionnelle** compléter ce projet avec différentes fonctionnalités.
   - Proposer plusieurs niveaux de difficultés dans lesquels les dimensions du terrain augmentent avec la difficulté et/ou la taille des cibles diminue avec la difficulté.
   - Stocker localement sur le navigateur la liste des meilleurs temps pour chaque nombre de cibles de départ. On peut alors utiliser la fonctionnalité "Web Storage". Après une recherche d'informations sur internet, utilisez l'objet *localStorage* et ses fonctions *setItem* et *getItem* pour mémoriser la liste de ces scores. A la fin de chaque partie on peut alors comparer le temps obtenu avec le meilleur score connu et le remplacer si nécessaire,

éventuellement après avoir demandé son nom au joueur. Vous pouvez dans ce cas compléter le document html avec une zone Hall of fame où sont affichés les meilleurs temps.

- Modifier le comportement du jeu pour que les cibles disparaissent (sans être comptabilisées) après un certain temps (éventuellement aléatoire). Evidemment il faut alors que lorsqu'une cible disparaît une autre apparaisse aléatoirement pour la remplacer.
- Si vous réalisez cette extension, créez un autre fichier de script différent de scripts/project201x.js et un autre document html différent de mini-projet-js-201x.html. Vous l'indiquerez dans le fichier lisezmoi.txt.
- Réaliser un second jeu dans lequel chaque cible a un certain nombre de points (éventuellement négatif). Dans ce cas, les cibles n'apparaissent pas toutes au début mais à des moments aléatoires et pour une durée qui peut être limitée. Le but est lors d'atteindre un certain nombre de points dans le plus petit temps. On peut ensuite imaginer d'autres variantes avec des cibles qui donnent des bonus ou malus en temps.
- Si vous réalisez cette extension, créez un autre fichier de script différent de scripts/project201x.js et un autre document html différent de mini-projet-js-201x.html. Vous l'indiquerez dans le fichier lisezmoi.txt.
- Vous pouvez imaginer vos propres extensions mais dans ce cas vous les présenterez précisément dans le fichier lisezmoi.txt.

9. Rendez votre travail sous la forme d'une archive dont la structure respectera celle de l'archive fournie. Le nom de cette archive sera *votreNom*-projetjs.zip. Cette archive contiendra un répertoire dont le nom sera *votre-nom*-projet. Le contenu de ce répertoire sera organisé ainsi :
   - un fichier lisezmoi.txt avec votre nom et prénom. Ce fichier mentionnera pour chacune des questions précédentes si elle a été traitée ou non et les éventuels problèmes de votre solution par rapport au cahier des charges demandé. Si vous avez ajouté des fonctionnalités, vous les détaillerez également dans ce fichier.
   - les fichiers mentionnés en introduction.

Tous vos fichiers seront codés en UTF-8. Les noms des fichiers (y compris leurs extensions) seront en minuscules.

PARTIE III. Rendu

Rendez votre travail sous la forme d'une archive dans Moodle.

Le nom de cette archive sera *votreNom*-projet.zip.

Cette archive contiendra un répertoire dont le nom sera *votre-nom*-projet.

Le contenu de ce répertoire sera organisé ainsi :

10. Un répertoire mini-projet-ocaml-202x avec dedans :
11. Les fichiers fournis que vous aurez complétés dans le répertoire `src/`
12. Les fichiers `AUTHORS` et `README` que vous aurez complétés. Le fichier `README` explique ce que vous avez fait, ce qui marche, les bugs connus, et ce que vous n'avez pas eu le temps de faire. Il n'y a pas forcément besoin de raconter sa vie dans ce fichier, c'est surtout pour pouvoir confronter ce que vous pensez de votre projet à ce qu'en disent les tests.
13. un répertoire `tests_perso/` pouvant contenir plusieurs sous-répertoires, dans lequel vous rangez tous les tests que vous aurez rédigés. Pour être valables, les tests doivent passer avec ma solution (ils peuvent échouer avec la vôtre, dans ce cas signalez-le dans le `README`). Si vos tests débusquent des erreurs dans le code d'autres groupes, vous pourrez gagner des bonus.
14. il n'est pas nécessaire de rendre le répertoire `test/` fourni. Si vous les rendez, le code qu'il contient ne sera pas évalué.
15. Dans le dossier mini-projet-js-202x, vous mettrez l'ensemble des fichiers (js, html, css…) au bon endroit.
16. Un fichier compte rendu du projet en pdf et avec son format source (doc, docx, odt au choix).

Tous vos fichiers seront codés en UTF-8. Les noms des fichiers (y compris leurs extensions) seront en minuscules.

PARTIE IV. Travail collaboratif

Vous devrez vous organiser pour vous partager le travail et collaborer efficacement en utilisant un dépôt Git.

Une phase d'analyse collective doit vous aider à choisir et préciser les formats que vous allez utiliser, le type de questions pris en compte et à découper votre travail sous la forme d'un Backlog de fonctionnalités que vous pourrez gérer vous-mêmes ou à l'aide d'un site comme https://framaboard.org/ ou https://trello.com/fr (Des outils installables comme https://kanboard.org/ ou https://www.openproject.org sont également disponibles.)

Vous aurez un dépôt par groupe et vous collaborerez grâce à lui. Vous pourrez le faire à un niveau élémentaire en pushant directement dans la branche master du dépôt.

Il faudra alors faire attention à travailler sur des fichiers différents ou éviter de provoquer des conflits.

Une façon plus satisfaisante de travailler sera de pusher vers des branches différentes et de les intégrer par des Pull Requests correspondant au développement d'une nouvelle fonctionnalité (feature) ou une correction de bugs.

Un membre du groupe pourra ainsi demander à un autre membre du groupe de passer son code en revue avant de l'intégrer.

Le travail est à rendre au plus tard pour le 10 juin 2022 sous la forme d'un projet versionné sur Github.

PARTIE V. Critère d'évaluation

Barème prévisionnel (à titre indicatif uniquement) :

Ocaml : Code	**/5**
JS : Un petit vaisseau spatial	**/8+**
Les fonctions sont courtes et bien nommées	/1
Commentaires là où c'est utile	/1
Structure du code	/2
Élégance du code	/2
Qualité du code	/2
Bonus	/1
Documentation	**/4**
Installation	/1
Lancement	/1
Description de l'interface utilisateur (avec un exemple)	/1
Description des fonctionnalités disponibles	/1
Versionnage et organisation du travail	**/3**
Git est utilisé	/1
tous les membres de l'équipe ont fait au moins 2 commits	/1
qualité des messages de commit	/1
TOTAL	**/20+**