

HTML5, web-workers : le monde parallèle du javascript

Par restimel



www.openclassrooms.com

*Licence Creative Commons 2.2.0
Dernière mise à jour le 15/06/2012*

Sommaire

Sommaire	2
Lire aussi	2
HTML5, web-workers : le monde parallèle du javascript	4
Partie 1 : La base des web-workers	5
Introduction à la programmation parallèle	5
La programmation séquentielle	5
La programmation impérative	5
La programmation événementielle	5
La programmation parallèle	6
Communication par mémoire partagée	7
Communication par passage de message	7
Les Dedicated Workers	8
Premiers pas	9
Mon premier Worker	9
La communication : Envoyer et recevoir des messages	10
Envoyer un message : postMessage()	10
Recevoir des messages : onmessage	11
Gérer un worker	12
Restrictions	12
L'espace global	12
Arrêter un worker	12
Les shared-workers	13
Premiers pas	14
La communication	14
Recevoir une nouvelle connexion : onconnect	14
Les ports	15
postMessage() et onmessage	15
Nommer vos Shared-workers	16
Ne plus utiliser un shared-worker	16
TP : Jeu des chiffres et des lettres	18
Présentation de l'exercice	18
Le jeu des chiffres	18
Le programme sans worker	18
À vous de jouer	24
Correction	24
Identifier les besoins	25
Instanciation du worker	25
La communication avec le Worker	25
Le worker	26
Cas particulier	27
Solution avec un worker	27
Partie 2 : Pour tout savoir sur les web-workers	36
Tout est communication	36
Le canal de communication	36
MessageChannel	36
Mieux communiquer	38
En savoir plus sur celui qui nous parle	38
Communiquer avec des objets	38
Terminer une communication	39
close() : arrêter la communication	39
Manipuler les workers	40
La gestion des erreurs	40
onerror : détecter une erreur	40
Débugger un worker	40
Connaître ses origines	41
location	41
navigator	42
Ajouter un script dynamiquement : importScripts	42
Enchaîner les workers	43
Créer un worker inline	44
Quand utiliser des web-workers ?	45
Quand ne pas utiliser les workers ?	46
Quand utiliser les workers ?	46
La performance	47
Le support des navigateurs	47
Tour d'horizon	48
Chrome / Safari (Webkit)	48
Firefox	48
Internet Explorer	49
Opera	49
TP : jeu du Gomoku	50
Présentation de l'exercice	50
Quelques explications	50

À vous de jouer	50
Correction	58
Explication de la mise en place d'un worker	58
Solution avec le worker	63
Pour aller plus loin	71
Exercice supplémentaire : traitement d'image	71
La marche à suivre	72
Solution sans worker	72
Solution avec worker	77



HTML5, web-workers : le monde parallèle du javascript



Le tutoriel que vous êtes en train de lire est en **bêta-test**. Son auteur souhaite que vous lui fassiez part de vos [commentaires](#) pour l'aider à l'améliorer avant sa publication officielle. Notez que le contenu n'a pas été validé par l'équipe éditoriale du Site du Zéro.



Par

[restimel](#)

Mise à jour : 15/06/2012

Difficulté : Difficile



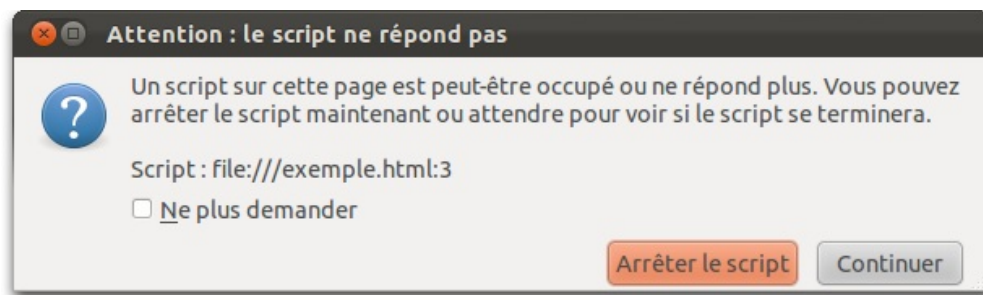
Dans ce qu'on nomme le HTML5, de nombreuses nouvelles fonctionnalités ont été ajoutées au javascript. Parmi celles-ci se trouvent les **web-workers** ou la possibilité d'exécuter du code en parallèle en javascript. Effectivement depuis sa création, le javascript souffre d'un défaut de taille : il est mono-thread !



Qui n'a jamais vu ses pages ne répondant plus pendant un certain temps ?

Qui n'a jamais vu ses animations saccader lorsqu'un autre script démarre ?

Qui n'a jamais vu son super code être coupé par une alerte invitant l'utilisateur à interrompre le script ?



Délai

d'exécution d'un script atteint.

Dans ce tutoriel, nous allons voir comment résoudre ces problèmes, en réalisant des exécutions de code en simultané.

Pour bien comprendre ce tutoriel, il est nécessaire d'avoir une connaissance solide du javascript. Avoir suivi et bien compris les trois premières parties [du tutoriel sur le javascript](#) est le strict minimum.

Le fait que ce tutoriel traite de fonctionnalités HTML5 implique que seuls les navigateurs récents (Firefox 3.5, Chrome 3, Opera 10.60, Internet Explorer 10, ...) pourront les supporter. Toutefois selon les navigateurs certaines fonctionnalités sont plus ou moins bien supportées, comme nous allons le voir dans la dernière partie de ce tutoriel.

Partie 1 : La base des web-workers

Cette partie va vous aider à découvrir et à prendre en main rapidement les workers.

Introduction à la programmation parallèle

Avant de plonger dans les web-workers, nous allons d'abord voir, et surtout comprendre, ce qu'est la programmation parallèle.

Le but d'un programme est d'exécuter une tâche. Pour réaliser celle-ci, on donne à l'ordinateur une liste d'instructions qu'il va effectuer. Il existe plusieurs manières de traiter ces instructions.

Ainsi nous allons voir ici, et surtout comprendre, différentes manières d'exécuter des instructions.

Il s'agit d'une partie théorique, mais rassurez-vous ça sera la seule de ce tutoriel.

La programmation séquentielle

La programmation impérative

Il s'agit sans doute de la méthode la plus naturelle, et il s'agit de la base de la programmation. Lorsque l'ordinateur a fini de traiter une instruction, il exécute la suivante.

Imaginez un jardinier. Dans son jardin, il fait pousser des plantes, les soigne et répond aux questions de quelques curieux. Ainsi s'il s'agissait d'un programme, nous lui donnerions les tâches suivantes :

- Arroser les plantes
- Soigner les plantes
- Répondre aux questions

Ainsi il commence par arroser les plantes. Dès qu'il a fini, il va les soigner. Et lorsqu'il a fini de les soigner, il va répondre aux questions des curieux.

Bien sûr, une liste d'instructions n'est pas forcément linéaire. On peut ajouter des structures de contrôle (comme les structures conditionnelles avec les **if**), des boucles (comme **for** ou **while**), ou des sous-ensembles d'instructions (comme les fonctions). Toutefois cela ne change pas le fait que le code reste séquentiel : dès qu'une instruction est terminée, on passe à la suivante.

Si j'écris ce code :

Code : JavaScript

```
var a = 1;
var b = 2;

function calcul1(c,d) {
    return c * d;
}

function calcul2(c, d) {
    return c + d;
}

a = calcul1(a, b);
b = calcul2(a, b);

alert("a=" + a + "\nb=" + b);
```



Qu'est-ce qui s'affiche avec ce code ?

Normalement, en lisant ce code, vous êtes capable de me dire le résultat de l'affichage. Car il suffit de partir du haut, de traiter les instructions une à une séquentiellement, et on finit par trouver le résultat.

Jusqu'ici, tout est clair ?

Cela vous paraît évident que le résultat d'un code est prévisible ? En fait ça ne l'est pas tant que ça ...

La programmation événementielle

Imaginez une page html, dans laquelle il y a trois boutons. Un bouton qui réalise le `calcul1`, un bouton pour le `calcul2` et un troisième pour l'affichage.

Cela pourrait donner cette page :

Code : HTML

```
<!doctype html>
<html lang="fr">
<head>
  <meta charset="utf-8">
  <title>Page avec 3 boutons</title>
</head>
```

```

<body>
  <button onclick="a = calcul1(a, b);">Calcul 1</button>
  <button onclick="b = calcul2(a, b);">Calcul 2</button>
  <br>
  <button onclick="alert('a=' + a + '\nb=' + b);">Alert</button>

  <script>
    var a = 1;
    var b = 2;

    function calcul1(c, d){
      return c * d;
    }

    function calcul2(c, d){
      return c + d;
    }
  </script>
</body>
</html>

```

Cela correspond au même code que précédemment. Sauf que si je pose la même question : qu'est-ce qui s'affiche avec ce code ? Vous voilà bien embêtés... Et oui ! Car le résultat dépend des actions de l'utilisateur avant qu'il demande l'affichage. Il peut aussi bien cliquer 2 fois sur le bouton `calcul2` puis sur le bouton `calcul1` et enfin demander l'affichage. Mais il peut très bien demander tout de suite l'affichage sans avoir réalisé aucun calcul. Il n'est pas possible d'anticiper le résultat.

C'est ce qu'on appelle la **programmation événementielle** : une action est déclenchée lors d'un événement. Dans notre code précédent, les événements déclencheurs sont les clics de la souris sur les boutons. Les fonctions ne seront exécutées que lorsque l'utilisateur aura cliqué sur un bouton.

Ce type de programmation rejoint la **programmation asynchrone**. Lorsqu'on exécute une tâche de manière asynchrone, son résultat n'est pas connu tout de suite. Pour obtenir son résultat, on écrit une fonction qui se déclenchera lorsque le résultat est connu. On voit bien qu'on a besoin d'un événement (quand le résultat est accessible) pour continuer l'exécution du code. Dans le cas d'une **programmation synchrone**, lorsqu'on exécute la fonction, le résultat est connu tout de suite (on ne continue pas tant que le résultat n'est pas connu).

Pour mettre en évidence ce caractère asynchrone et événementiel dans la programmation en javascript, étudions ensemble un cas pratique : le chargement d'une image.

Si vous souhaitez afficher une nouvelle image et récupérer ses dimensions, qu'allez-vous faire ?

Si vous écrivez `var img = new Image(); img.src = cheminImage;`, vous ne pourrez pas obtenir ses dimensions tout de suite. En effet l'objet image va charger l'image de manière asynchrone. Ainsi pour obtenir ses dimensions, il va falloir utiliser l'événement `onload` pour lire les informations lorsque l'image est chargée.



Mais même si les séquences sont exécutées de manière asynchrone, à chaque fois il n'y en a qu'une qui travaille. Les autres séquences attendent que la séquence active se termine pour remplir leur tâche.

Reprenons l'analogie du jardinier, et rendons-le un peu plus intelligent :

- Il ne va arroser les plantes que lorsqu'elles manquent d'eau.
- Il ne va soigner les plantes que lorsqu'il observera des parasites.
- Il ne répondra aux curieux que lorsqu'ils auront une question à poser.

Donc dès que les plantes sont envahies par des parasites, le jardinier se met alors à la tâche et s'en va soigner ses plantes.

Imaginons, alors, que pendant son travail, les deux autres événements apparaissent : les plantes manquent d'eau, et un curieux désire poser une question.

Notre jardinier termine alors son travail de soin, s'en va chercher de l'eau, les arrose. Puis, enfin, va rejoindre le curieux pour répondre à sa question. On peut alors facilement estimer que dans ce cas de figure, le curieux peut avoir passé beaucoup de temps à attendre. Et s'il s'agit d'une personne importante, comme un client, il serait dommage de le vexer en le faisant attendre aussi longtemps. 😞

Malheureusement en javascript, il n'est pas possible d'ordonnancer les tâches qui ont été déclenchées en fonction de leur priorité. Elles seront exécutées dans l'ordre de déclenchement. On pourrait essayer de le faire manuellement. Ainsi au début d'une nouvelle tâche, on vérifie s'il ne faut pas en exécuter d'abord une autre plus prioritaire, et remettre celle en cours à plus tard. Non seulement c'est compliqué à mettre en place, mais on a toujours un problème : si la tâche en cours au moment du déclenchement de l'événement est très longue, notre important client devra toujours patienter jusqu'à la fin de celle-ci.



Mais alors comment faire ?

Il suffit d'embaucher une nouvelle personne 😊

La programmation parallèle

La **programmation parallèle** consiste à exécuter plusieurs tâches en même temps !

Ça ne semble pas difficile, n'est-ce pas ? Et pourtant c'est loin d'être simple !

À la question précédente, certains petits malins ont dû penser : il suffit que le jardinier arrose ses plantes et réponde en criant



Oui, mais le jardinier doit quand même faire deux actions en même temps : arroser et crier. Cela est possible, car l'humain est une machine formidable capable de réaliser de très nombreuses choses en même temps.

Cependant pour un ordinateur, ce n'est pas aussi simple. Car initialement, les ordinateurs sont conçus pour traiter les informations séquentiellement : le processeur exécute les instructions les unes à la suite des autres. Dans ce contexte, il est difficile d'imaginer de pouvoir réaliser plusieurs actions en même temps.

Et pourtant, quand vous utilisez votre ordinateur et que vous copiez un (gros) fichier, vous pouvez toujours continuer à naviguer dans vos répertoires. Vous n'êtes pas (plus) obligé d'attendre la fin de la copie.

Ceci est possible, car votre système d'exploitation gère ce qu'on appelle un **scheduler** (ordonnanceur en français). Il va allouer un certain temps d'exécution à chaque code qui désire fonctionner à cet instant. Ainsi, il va exécuter un premier code (comme la copie du fichier), puis l'interrompre pour démarrer un deuxième code (comme la navigation dans les répertoires), puis interrompre ce dernier pour relancer le premier là où il en était précédemment ... et alterner ainsi de suite avec tous les programmes qui tourment à ce moment.

Bien que chaque code s'exécute à tour de rôle, il s'agit bien d'actions simultanées puisque lorsque la première tâche est en cours d'exécution, la deuxième est également en cours d'exécution.

Depuis les années 2000, les ordinateurs multiprocesseurs ou les processeurs multicœurs se sont généralisés. Aujourd'hui, pratiquement tout le monde possède un ordinateur disposant de cette technologie. Or cette technologie permet de réellement exécuter plusieurs instructions en même temps (1 par cœur ou par processeur).

Cependant, pour qu'un programme puisse bénéficier de ces avantages, et donc exécuter deux codes (ou plus) de manière concurrente, il doit indiquer qu'un certain code ne s'exécute plus dans la même pile que le code principal.

Une pile, ou plutôt **pile sémantique** (pour ne pas la confondre avec la pile d'exécution qui liste les fonctions actives du programme, et même si dans chaque pile sémantique se trouve une pile d'exécution) est un ensemble d'instructions que l'ordinateur doit exécuter.

Il est possible que vous entendiez les termes **thread** ou **processus**.

Un **thread** (ou fil d'exécution en français) contient une pile sémantique et se charge de l'exécuter.



Un **processus** (ou process en anglais) contient un ou plusieurs threads. Le scheduler se chargera de séquencer ces processus afin qu'ils puissent tous fonctionner en même temps.

Les threads à l'intérieur du même processus peuvent partager une mémoire commune alors que les processus ne partagent pas de mémoire entre eux.

En javascript, c'est au moteur de choisir s'il crée un nouveau thread dans le même processus, ou s'il crée un nouveau processus (avec un thread dedans bien sûr). Le programmeur ne peut pas choisir spécifiquement l'un ou l'autre.

Dans la suite de ce tutoriel, j'utiliserai le terme **thread** pour désigner ce qui est en fait une pile sémantique, car ce terme est très généralement utilisé sur le web sans faire la distinction entre ces différents concepts. Comme nous ne gérons pas le programme dans le système d'exploitation, nous n'aurons donc pas de quiproquo.

En programmation parallèle, il est très souvent nécessaire d'avoir un système de communication entre les threads. Il existe deux principaux modèles de communication : le modèle par mémoire partagée, et le modèle par passage de messages.

Communication par mémoire partagée

Dans un modèle utilisant la mémoire partagée (**shared memory** en anglais), chaque thread exécute sa propre séquence d'instructions, mais ils peuvent accéder à des données communes.

En gros, cela signifie qu'il existe une zone mémoire où sont stockées des variables utilisées par plusieurs threads.

Ce système présente des inconvénients très délicats, comme l'**accès concurrent** à la même variable, les **deadlocks**, etc...

Accès concurrent : Plusieurs threads accèdent aux mêmes ressources en même temps. Par exemple : Deux threads veulent incrémenter un compteur initialisé à 0. Si les deux threads y accèdent en même temps, ils lisent tous les deux 0, ajoutent 1 et enregistrent la modification. La valeur du compteur sera alors 1 au lieu de 2, car un thread aura écrasé la valeur enregistrée par l'autre thread.



Les deadlocks : C'est un problème qui peut survenir quand plusieurs threads désirent accéder à plusieurs ressources partagées en même temps. Par exemple : Un thread bloque une ressource partagée (afin d'éviter un problème d'accès concurrent) et désire accéder à une autre ressource bloquée par un autre thread qui désire accéder à la ressource bloquée par le premier. Résultat : tout le monde attend et le programme est bloqué !

Désolé, je ne vais pas détailler ces problèmes plus que ça, car, en javascript, c'est l'autre modèle qui est utilisé ! Si le sujet vous intéresse, je vous invite à consulter de nombreux sites dédiés à ce sujet.

Communication par passage de message

Dans un modèle "passage de message" (**message passing** en anglais), contrairement au modèle précédent, chaque thread est complètement distinct des autres. Ils ne partagent rien. Le seul lien qu'ils peuvent avoir est un canal de communication afin qu'ils puissent s'envoyer des messages.

Pour reprendre l'analogie du jardinier : nous désirons maintenant satisfaire au mieux les clients, et donc ne pas les faire patienter inutilement. Ainsi nous avons maintenant deux personnes : un jardinier qui s'occupe des plantes et un commercial qui s'occupe des clients. Le commercial ne va pas au jardin, et le jardinier ne se rend pas à la boutique. Mais régulièrement le jardinier envoie des messages au commercial pour lui donner un état des lieux. Et le commercial lui envoie pour lui demander de préparer telle ou telle plante.

Le principe du "message passing" c'est ça : **on ne touche pas aux affaires des autres !** (même pas avec les yeux)

Comme il existe de nombreuses variantes, je vais me contenter de vous parler de celle qui concerne le javascript : le passage par message asynchrone, et on pourrait ajouter qu'il s'agit d'un dialogue.



Qu'est-ce que je veux dire par là avec tous ces mots compliqués ?

Analysons-les un par un.

Dialogue : La communication ne se fait qu'entre deux tâches. On aurait pu imaginer envoyer un message à tout le monde et que seul celui que ça concerne le lit (c'est ce qu'on fait lorsqu'on crie à son copain qui est de l'autre côté de la rue : tous les passants reçoivent aussi le message). Mais là, non ! Il n'y a que le destinataire du message qui pourra le lire.

Un dialogue signifie aussi que les deux peuvent initier un message. Une tâche n'est pas obligée d'attendre que l'autre lui parle pour lui envoyer un message. Sinon ça serait un monologue. 😊

Asynchrone : Lorsqu'une tâche envoie son message, elle n'attend pas de réponse. C'est-à-dire que dès qu'elle a envoyé son message, elle peut s'occuper des instructions suivantes. Cela ne veut pas dire qu'il n'y aura pas de réponse, mais juste qu'elle pourra arriver (beaucoup) plus tard.

Si vous appelez une amie au téléphone, votre communication sera synchrone car lorsque vous posez une question, vous attendez votre réponse et vous ne faites rien d'autre en attendant.

Si par contre vous lui écrivez, votre communication est asynchrone. Car lorsque vous avez envoyé votre lettre, vous n'attendez pas devant la poste sa réponse.

Passage par message : ben ... euh, on communique avec des messages. 🤖

Un petit résumé de ce qu'il faut bien retenir de ce principe :

- Les messages ne sont envoyés qu'à un seul autre thread.
- L'envoi d'un message n'est pas bloquant.
- La lecture des messages est événementielle, ce qui peut mener à avoir plusieurs messages en attente de lecture : ils sont en file d'attente (**message queue** en anglais).
- Et surtout on ne sait rien de ce qu'il se passe chez son voisin mis à part ce qu'il nous dit dans ses messages.

Et voilà la partie théorique est terminée ! Vous pouvez vous réveiller !

Il y avait plein de notions à comprendre, mais maintenant rassurez-vous, nous allons pouvoir passer à la pratique.

Et grâce à toutes ces notions, vous allez surtout bien comprendre tout ce que vous allez faire avec les web-workers (et aussi ce que vous ne pouvez pas faire).

Les Dedicated Workers

Maintenant qu'on a rapidement vu en quoi consistait la programmation concurrentielle, nous allons entrer dans le vif du sujet. Nous allons découvrir dans ce chapitre comment créer un code s'exécutant en parallèle en javascript.

Ne vous en faites pas, vous serez bien guidé 😊

Premiers pas

Il existe deux types de **web-workers**. Le premier que nous allons analyser ici correspond aux **dedicated-workers** (workers dédiés). Ces workers, une fois créés, sont liés à leur créateur et uniquement à celui-ci.

Mon premier Worker

Pour commencer, nous allons tout de suite voir à quoi ressemble un dedicated-worker.

Nous allons créer un jardinier, qui arrosera continuellement ses plantes, et les récoltera quand elles auront bien grandi. Et nous afficherons le nombre de plantes qui ont été récoltées.

Nous avons donc deux fichiers : un qui est directement inclus dans la page web (que j'ai appelé main.js) qui sert à afficher le nombre de plantes récoltées ; et un qui correspond au code devant s'exécuter en parallèle (que j'ai nommé worker.js) qui sert à simuler le potager du jardinier.

main.js	worker.js
<p>Code : JavaScript</p> <pre>//Création d'une zone d'affichage var elem=document.createElement("output"); elem.value = "Le jardinier marche"+ " vers le potager"; document.body.appendChild(elem); //Création d'un worker if(window.Worker){ //le navigateur supporte les workers var worker=new Worker("worker.js"); worker.onmessage=function(event){ elem.value = event.data; }; }else{ //le navigateur ne supporte pas les workers alert("Désolé votre navigateur "+" "ne supporte pas les workers ! ☹️"); }</pre>	<p>Code : JavaScript</p> <pre>setTimeout(function(){ //le setTimeout ne sert qu'à permettre //de bien voir les premiers messages postMessage("Le jardinier est dans son potager"+ " et est en train de faire pousser ses "+" "premières plantes"); var jardin = [0,0,0,0,0]; var plantePrete = 0; var parcelle; while(true){ //boucle infinie, on arrose toujours les plantes for(parcelle = 0; parcelle < 5; parcelle++){ //on arrose un peu, et ça fait grandir la plante jardin[parcelle] += Math.random(); if(jardin[parcelle] > 1000000){ //la plante est suffisamment grande et on la cueille jardin[parcelle] = 0; plantePrete++; postMessage("Le jardinier a pu récolter "+" plantePrete +" plantes"); } }//boucle de fin de la parcelle }//boucle infinie },1000);</pre>

Essayer !



Si vous testez ce code avec le moteur Webkit (qui est utilisé par Chrome et Safari), vous devez mettre la page en ligne. Car, comme nous le verrons à la fin de ce tutoriel, webkit refuse d'utiliser les web-workers en local !

Qu'observons-nous ? Juste un texte indiquant le nombre de plantes récoltées, et ce nombre qui grandit. Pas grand-chose d'impressionnant, certes ! Mais en analysant le code, vous devriez remarquer quelques subtilités.

Que pouvons-nous observer d'intéressant dans le fichier `main.js` ?

- À la ligne 13 : `new Worker("worker.js")`. C'est la création et le démarrage d'un nouveau worker. Le paramètre correspond à l'URL du code javascript à exécuter dans ce worker.
- À la ligne 14 : `worker.onmessage`, il s'agit d'un listener appliqué sur le worker. Nous reviendrons là-dessus dans peu de temps.
- À la ligne 15 : `event.data`, il s'agit du message reçu. Nous reviendrons aussi là-dessus.

Pour le fichier `worker.js`, il y a deux choses à remarquer :

- Ligne 5 et ligne 25 : `postMessage()`, il s'agit d'une fonction permettant d'envoyer des messages au thread principal. Nous y reviendrons dans peu de temps.
- Ligne 13 : `while(true)` ! Une boucle infinie 😞 ! Vous ne rêvez pas, à la ligne 8, c'est bien le commencement d'une boucle qui ne se termine jamais !
Alors non, dans un worker il n'est pas nécessaire d'écrire une boucle infinie, c'est juste que dans cet exemple je voulais écrire une tâche qui ne se finissait jamais.

Et pour les plus perspicaces, cela signifie bien que dans ce thread, l'interpréteur javascript ne rend jamais la main. Or vous avez sans doute remarqué que le texte qui s'affichait correspond à celui de la ligne 25, et vous devriez savoir que d'habitude les modifications du DOM ne se font que lorsque l'interpréteur rend la main. Oui mais là, nous avons 2 threads, un qui garde toujours la main et l'autre qui gère l'affichage (et dans cet exemple, il est plutôt fainéant). Cela n'aurait donc pas été faisable sans la parallélisation.



Alors que se passe-t-il réellement dans cette page ?

- Au commencement était la page HTML
- La page charge le fichier `main.js`
- Une balise `output` est créée pour gérer l'affichage
- On crée un worker :
 - Le fichier `worker.js` est chargé dans un environnement qui tourne en tâche de fond (qu'on nomme souvent `background`), il s'agit d'un thread différent du thread principal.
 - Un message est envoyé à `main.js` pour lui dire que le jardinier travaille
 - On définit le jardin
 - Dans la boucle infinie, pour chaque parcelle on fait pousser la plante
 - Lorsqu'une plante est cueillie, on envoie un message indiquant le nombre de plantes cueillies.
- Quand un message du worker est reçu, on l'affiche

Et voilà vous avez réalisé votre premier programme javascript à exécution parallèle 😊



Vous avez pu remarquer qu'à la ligne 10, la condition vérifie que le navigateur supporte l'objet `Worker`. Cela est fortement conseillé pour garder une compatibilité avec les vieux navigateurs et leur proposer une autre solution. Afin d'éviter de trop surcharger les codes qui suivent, je n'ajouterai plus ce test. Cependant, cela ne vous dispense pas de le faire quand même 😊

La communication : Envoyer et recevoir des messages

Je vais me répéter, mais pour bien comprendre le modèle de passage de messages, il faut garder en tête que chaque thread travaille dans son coin. Et lorsque un thread désire donner un ordre ou une information à un autre thread, il va devoir lui envoyer un message.

Pour cela il devra prendre une lettre, écrire son message, et l'envoyer par la poste. Son destinataire n'a plus qu'à lire le message et lui répondre par le même intermédiaire.

Vous croyez que je plaisante ? 😊 En fait pas du tout, c'est exactement ça ! 😊 Bien sûr la lettre n'est pas faite en papier, et la poste est électronique, mais le principe est exactement celui-ci.

Les plus perspicaces auront vite remarqué un petit problème dans cette méthode : pour avoir une réponse, il faut attendre. Il faut attendre que son interlocuteur prenne le temps de lire la réponse puis envoie à son tour un message. La communication est asynchrone. Toutefois cela ne devrait pas vous inquiéter, l'AJAX fonctionne exactement de la même manière.

Envoyer un message : `postMessage()`

Pour envoyer un message, il suffit d'utiliser `postMessage`.

Cette fonction possède un paramètre de type string : c'est le message à envoyer !

Il s'agit d'une méthode de l'objet `worker`.

Code : JavaScript

```
var w = new Worker("cheminDuWorker.js"); //création du dedicated-
worker
w.postMessage("un message"); //on envoie "un message" au worker w
```

Depuis le worker c'est encore plus simple, il suffit d'utiliser directement la fonction `postMessage`.

Code : JavaScript

```
postMessage("un message depuis le worker"); //le worker envoie son
message au thread principal
```

Recevoir des messages : onmessage

Envoyer des messages, c'est bien, mais il faut aussi pouvoir les lire 😊

On peut le faire grâce au listener `onmessage`. Comme c'est un listener (comme onclick, onload, etc...) il va falloir lui passer une fonction callback qui sera exécutée lors de son déclenchement.

Pour lire le message, il suffira d'accéder à la propriété `data` de l'événement.

Vous êtes maintenant capable d'écrire un petit échange avec le worker. En voici un par exemple :

main.js	worker.js
<p>Code : JavaScript</p> <pre>//création du worker var w = new Worker("worker.js"); //réception des messages w.onmessage = function(event){ alert("Le worker a répondu :\n" + event.data); if(event.data.substr(0,7) == "Bonjour"){ //Si le message commence par "Bonjour" var nom=prompt("Quel est ton nom ?"); w.postMessage("L'utilisateur s'appelle : "+nom); } }; w.postMessage("Bonjour");</pre>	<p>Code : JavaScript</p> <pre>//réception d'un message onmessage=function(event){ if(event.data == "Bonjour"){ postMessage("Bonjour, je suis un worker"); }else{ postMessage("Maintenant je sais que tu es "+ event.data.substr(26) +" !"); } };</pre>
Essayer !	

On commence par créer un worker, on met en place un listener puis on lui envoie "Bonjour".

Du côté du worker, il n'y a qu'un listener qui répond au message reçu.



Dans de très nombreuses fonctions liées aux listeners, vous devez être habitué à voir une ligne du style : `event=event||window.event;` Cette ligne sert à assurer la compatibilité avec IE. Toutefois, comme les workers ne fonctionnent qu'à partir de IE10 et que celui-ci supporte la gestion des événements préconisée par le DOM, il n'est pas nécessaire d'ajouter cette ligne. 😊



Est-ce qu'il est possible d'utiliser `addEventListener` ?

Bien sûr, il s'agit d'un événement comme un autre, il est donc possible d'ajouter des listeners. Ainsi les codes précédents donneraient :

main.js	worker.js
<p>Code : JavaScript</p> <pre>//création du worker var w=new Worker("worker.js"); //ajout d'un listener pour réception d'un message w.addEventListener("message", function(event){ alert("Le worker a répondu :\n" + event.data);</pre>	<p>Code : JavaScript</p> <pre>function reponse(event){ if(event.data == "Bonjour"){ postMessage("Bonjour, je suis un worker"); }else{</pre>

<pre> reponse : \n + event.data;; if(event.data.substr(0,7) == "Bonjour"){ var nom=prompt("Quel est ton nom ?"); w.postMessage("L'utilisateur s'appelle : "+nom); },false); w.postMessage("Bonjour"); </pre>	<pre> postMessage("Maintenant je sais que tu es "+ event.data.substr(26) + " !"); } //ajout d'un listener addEventListener("message", reponse,false); </pre>
Essayer!	

Gérer un worker Restrictions

Comme je l'ai déjà répété plusieurs fois, le worker travaille dans un espace qui lui est dédié et qui est complètement séparé de l'environnement d'origine. Cela a pour conséquence de ne pouvoir accéder à de nombreuses ressources comme l'objet `document` et l'objet `window`. Cela signifie donc que les variables globales dans la tâche principale ne sont pas accessibles dans le worker. Cela signifie aussi que la page (et le DOM) n'est pas accessible. Il n'est donc pas possible de manipuler ces éléments depuis le worker.

Ne désespérez pas ! Car en plus de l'utilisation des objets de bases comme `Math`, `Date`, `XMLHttpRequest`,... les workers disposent aussi de quelques outils permettant de compenser l'absence d'accès aux propriétés de l'objet `window`. Nous verrons cela dans le chapitre avancé, pour l'instant nous allons nous contenter des bases.

L'espace global

L'espace global est l'espace de nom principal qui contient tous les autres espaces de noms qui sont utilisés.

L'espace global est à l'intérieur du worker, il ne peut pas accéder à l'extérieur et il n'est pas accessible depuis l'extérieur. À l'intérieur du worker, on peut y faire référence, non pas avec `window`, mais grâce à la variable `self`.

Code : JavaScript

```

var toto = 1;
var tata = self.toto; //vaut 1

```

Arrêter un worker

Jusqu'à présent, nous avons créé et fait vivre un worker. Maintenant on va voir comment le tuer.



Pourquoi avons-nous besoin de tuer un worker ?

La raison principale est d'économiser des ressources. Car un worker consomme de la mémoire et du CPU. Si vous n'avez plus besoin de ses services, alors il est intéressant de libérer les ressources consommées. D'autant plus que certains navigateurs, comme Firefox, gèrent très mal la consommation excessive de mémoire. Le worker devient alors inaccessible mais existe toujours (et donc la mémoire est toujours consommée).

Tout d'abord on peut remarquer que désassigner la variable qui a servi à créer le worker ne suffit pas à l'arrêter. Prenons ces codes :

main.js	worker.js
<p>Code : JavaScript</p> <pre> // initialisation du worker var w = new Worker("worker.js"); delete w; </pre>	<p>Code : JavaScript</p> <pre> var l = 50000000; var tab = []; //on remplit un tableau for(var i = 0; i < l; i++){ //À chaque itération la mémoire consommée augmente tab.push("message"); } </pre>

Afin de constater l'état de la mémoire lors de l'exécution de ce code, vous pouvez ouvrir le "Gestionnaire des tâches"/onglet "Performances" (sous Windows) ou le "Moniteur système"/onglet "Ressources" (sous Linux). Les graphiques que vous pouvez observer indiquent le pourcentage de capacités du processeur et la quantité de mémoire vive (RAM) utilisés par les programmes

en cours d'exécution.

Vous remarquerez le taux d'utilisation de la mémoire qui continue de monter, alors que juste après sa création, à la ligne 3, nous n'avons plus aucune référence au worker. Le garbage collector ne s'occupe pas du worker car il s'agit d'un script indépendant. Il faut donc explicitement arrêter ce script.

Vous avez pu remarquer que lorsque la boucle s'est terminée, la mémoire est restée stable. Elle n'a pas diminué. Car même si le script n'a plus de code à exécuter, il n'est pas fini pour autant. Donc le tableau reste dans l'environnement global du script et donc en mémoire.



Si vous générez une erreur dans le worker, vous verrez que cela revient au même. En effet, lorsqu'une erreur survient, l'interpréteur s'arrête, mais le script vit toujours. Il est toujours possible de lancer d'autres fonctions.

On peut toujours fermer la page, mais c'est un peu violent quand même. 🤔

Il existe donc des méthodes propres aux workers pour les arrêter.

Éliminer un worker : `terminate()`

La fonction la plus efficace est `terminate` qui permet d'arrêter brutalement le worker. Elle s'applique sur le worker : `worker.terminate()`

Lorsqu'on fait appel à cette méthode, le worker arrête immédiatement sa tâche en cours puis libère les ressources dont il avait besoin.

main.js	worker.js
<p>Code : JavaScript</p> <pre>//préparation pour l'affichage var elem = document.createElement("output"); elem.value = "en cours de création"; document.body.appendChild(elem); // initialisation du worker var w=new Worker("worker.js"); //ajout d'un listener w.addEventListener("message", function(event){ elem.value="Le worker a déjà fait "+ event.data+" tours"; }); setTimeout(function(){ // Au bout d'une seconde, on arrête le worker w.terminate(); document.body.appendChild(document.createTextNode("Worker Éliminé")); },1000);</pre>	<p>Code : JavaScript</p> <pre>var l = 50000000; var tab = []; //on remplit un tableau for(var i = 0; i < l; i++){ //À chaque itération la mémoire consommée augmente tab.push("message"); //tous les 100 tours, on informe le thread principal if(i%100 === 0){ postMessage(i); } }</pre>
Essayer!	

Vous pouvez maintenant voir que la mémoire consommée augmente pendant une seconde puis revient à son niveau initial. Car cette fois le worker a été éliminé. 🤖

Désormais si vous essayez de lui envoyer un message, vous rencontrerez une erreur car le worker n'existe plus.



Si vous voulez relancer le worker, il faut en créer un nouveau ! Il n'est pas possible de le relancer là où il en était.

Maintenant vous pouvez déjà utiliser les dedicated-workers pour embellir vos programmes.

Dans la partie suivante, vous découvrirez un deuxième type de worker: le shared-worker.

Les shared-workers

Dans ce chapitre, nous allons découvrir un deuxième type de web-worker : les shared-worker.

Nous allons voir quels sont les différences avec les dedicated-worker. Nous allons y aller pas à pas afin de les manipuler sans crainte 😊

Premiers pas

Nous avons vu dans le précédent chapitre comment manipuler un worker et plus particulièrement les dedicated-workers.

Nous allons maintenant voir un deuxième type de workers disponible en javascript : les **shared-workers** (workers partagés en français). Contrairement aux dedicated-workers, ils peuvent être contrôlés par plusieurs threads. Il est donc possible de créer plusieurs références vers le même worker, et ces références peuvent même provenir de pages différentes.

Un exemple vaudra sans doute bien mieux qu'une longue explication.

main.js	worker.js
<p>Code : JavaScript</p> <pre>//Création d'une zone d'affichage var elem = document.createElement("output"); elem.value = "Log :"; document.body.appendChild(elem); if(window.SharedWorker){ //le navigateur supporte les shared-workers var w = new SharedWorker("worker.js"); w.port.onmessage = function(e){ elem.innerHTML += "
" + e.data; }; w.port.postMessage("Bonjour"); }else{ elem.textContent="Votre navigateur ne "+ "supporte pas les shared- workers ☹️"; }</pre>	<p>Code : JavaScript</p> <pre>var num = 0; //nouvelle connexion onconnect=function(e){ var port = e.ports[0]; port.postMessage("pret #" + num); num++; //réception des messages port.onmessage = function(event){ port.postMessage("réponse à "+ event.data); }; };</pre>
<p>Essayer à partir de la page 1 ! Essayer à partir de la page 2 !</p>	

À première vue, en essayant ce code, vous vous dites qu'il n'y a rien de bien exceptionnel.

Essayez maintenant d'ouvrir la page contenant ce code une deuxième fois (donc cette page sera ouverte dans deux onglets ou deux fenêtres). Qu'observez-vous ?

L'affichage de la page est différent alors qu'il s'agit du même code qui est exécuté 😊 !

Et si vous l'ouvrez une troisième fois, un #3 apparaît alors. Et le numéro s'incrémente pour chaque nouvelle page. Ou plus précisément pour chaque nouvelle connexion au worker.

Car contrairement aux dedicated-workers, celui-ci est partagé entre toutes ces pages. C'est-à-dire que ces pages communiquent avec le même thread, et num s'incrémente à chaque nouvelle connexion.

Et lorsqu'on ouvre ce code depuis une autre page, c'est toujours la même chose ! Ainsi il est possible de réaliser une communication entre deux pages différentes issues du même site.

Toutefois il n'est pas possible de communiquer avec un autre utilisateur. Les workers, étant des composants javascript, sont toujours exécutés "côté client", c'est-à-dire au niveau du navigateur d'un utilisateur. Les workers possèdent aussi une restriction : ces pages doivent provenir du même domaine(same origin policy).

Le shared-worker sera éliminé dès qu'il n'y a plus aucune référence à celui-ci. Ainsi si vous fermez toutes vos pages, et que vous en rouvrez une, le worker repartira de 0.

Le principe d'utilisation reste le même que pour les dedicated-workers et s'utilise de la même manière. Cependant vous avez pu remarquer que l'appel des méthodes diffère légèrement. Nous allons voir comment appréhender tout ça 😊

La communication

Le concept est toujours le même, les objets appartenant au worker ne sont accessibles que par lui, et lui ne peut pas accéder aux objets extérieurs. Leur seul lien reste toujours un canal de communication.

Recevoir une nouvelle connexion : onconnect

La première nouveauté par rapport aux dedicated-workers est le listener onconnect.

Ce listener est appelé à chaque fois qu'un code fait appel à ce worker. Il est donc particulièrement pratique pour adapter son code au nouveau venu (et surtout communiquer avec lui).

Dans l'événement généré, il y a une propriété qui va particulièrement nous intéresser. Il s'agit de **ports** qui est une liste d'objets de type *port*.

Avant d'aller plus loin, nous allons regarder ce qu'est un port.

Non ce n'est pas là où on abrite les bateaux. 😊

Les ports

Un **port logiciel** est un point de contact qui permet de communiquer avec un interlocuteur. Cela permet de bien distinguer tous les interlocuteurs puisqu'ils seront associés chacun à un port différent.

On peut considérer les ports comme des portes. Quand vous ouvrez une porte vous pouvez accéder à une autre pièce. Si vous désirez atteindre une troisième pièce, vous allez utiliser une porte différente. Si vous voulez que deux pièces de votre maison communiquent ensemble, il vous faudra une porte entre elles.

En informatique, c'est un peu la même chose ! Les ports vont nous servir à communiquer entre deux threads.

Ainsi quand on crée un shared-worker, deux ports sont aussi créés :

- L'un est attaché à l'objet shared-worker créé et se nomme `port`.
- L'autre est envoyé au thread du worker via l'événement `connect`, et est accessible via le premier élément de `ports`

Dans le code d'introduction, vous pouvez remarquer qu'à la ligne 4 du worker, je récupère explicitement ce port. Et dans le code de `main.js`, je fais appel à son frère en utilisant la syntaxe `w.port`.

postMessage() et onmessage

`postMessage()` et `onmessage` s'utilisent exactement de la même manière qu'avec les `dedicated-workers`. La seule différence est qu'ils sont associés à un port.

En réalité, avec les `dedicated-workers` il y a aussi des ports, mais ils sont intégrés à l'objet et à l'espace global. Du coup il n'est pas nécessaire d'y faire appel explicitement. Cela se justifie facilement puisqu'un `dedicated-worker` est dédié à celui qui l'a créé et donc ne communique qu'avec lui.

Il en va autrement des `shared-workers` qui peuvent avoir une multitude de correspondants. Il est donc nécessaire de bien préciser avec lequel on souhaite correspondre.



J'ai voulu utiliser `addEventListener` mais ça ne marche pas ! Pourquoi ?

Effectivement, il s'agit de la principale différence avec les `dedicated-workers`. Sur un port, si on souhaite lire les messages en attente il faut explicitement démarrer la distribution.

Ce démarrage se fait grâce à la fonction `start()`.

main.js	worker.js
<p>Code : JavaScript</p> <pre>//Création d'une zone d'affichage var elem = document.createElement("output"); elem.textContent = "Log :"; document.body.appendChild(elem); //Création du worker var w = new SharedWorker("worker.js"); w.port.addEventListener("message", function(e) { elem.innerHTML += "
"+e.data; }, false); w.port.start(); /* La ligne précédente est très importante pour démarrer la réception des messages sur ce port */ w.port.postMessage("Bonjour");</pre>	<p>Code : JavaScript</p> <pre>var num = 0; //ajout d'un listener de connexion addEventListener("connect", function(e) { var port = e.ports[0]; port.postMessage("Pret #" + num); num++; //on écoute sur le port port.addEventListener("message", function(e) { port.postMessage("Réponse à "+e.data); }, false); /* Pour démarrer la réception des messages sur ce port */ port.start(); }, false);</pre>

Essayer !



Il n'est pas nécessaire d'utiliser `port.start()` lorsqu'on utilise `port.onmessage`, car avec cette méthode il n'y a pas de "distribution" puisqu'il s'agit du seul endroit où est réceptionné le message. Le démarrage est donc réalisé de manière implicite.

Nommer vos Shared-workers

Si dans un code on veut créer deux shared-workers distincts à partir du même fichier, on ne pourra pas s'y prendre de cette façon :

Code : JavaScript

```
var w1 = new SharedWorker("toto.js");
var w2 = new SharedWorker("toto.js");
```

Puisque w1 et w2 feront appel au même worker partagé.

Le constructeur `SharedWorker` possède un deuxième argument qui permet de nommer un shared-worker.

Si maintenant je fais :

Code : JavaScript

```
var w1 = new SharedWorker("toto.js", "toto");
var w2 = new SharedWorker("toto.js", "tata");
```

Je possède désormais deux workers distincts car ils ne possèdent pas le même nom. Il est aussi possible d'utiliser ce nom pour faire explicitement appel à un worker existant.

main.js	worker.js
<p>Code : JavaScript</p> <pre>//Création d'une zone d'affichage var elem = document.createElement("div"); elem.textContent = "Log :"; document.body.appendChild(elem); //Création de workers var w1=new SharedWorker("worker.js", "monWorker"); w1.port.onmessage = function(e) { elem.innerHTML += "
W1 " + e.data; }; var w2=new SharedWorker("worker.js", "worker2"); w2.port.onmessage = function(e) { elem.innerHTML += "
W2 " + e.data; }; var w3=new SharedWorker("worker.js", "monWorker"); w3.port.onmessage = function(e) { elem.innerHTML += "
W3 "+e.data; }; w1.port.postMessage("Bonjour"); w2.port.postMessage("Bonjour"); w3.port.postMessage("Bonjour");</pre>	<p>Code : JavaScript</p> <pre>var num = 0; onconnect = function(e) { var port = e.ports[0]; port.postMessage("pret #" + num + " ? " + self.name); num++; port.onmessage=function(event) { port.postMessage("réponse à " + event.data); }; };</pre>

Essayer !

w1 et w2 ne partagent pas le même worker. Par contre w1 et w3 partagent le même worker. Car ils ont utilisé le même nom.

Si on crée un shared-worker à partir d'un nom qui existe déjà et d'une url différente de celle utilisée, alors une erreur apparaît.



Au passage, vous avez pu remarquer que `self.name` permet de connaître le nom attribué à ce worker.

Ne plus utiliser un shared-worker

Il est bien sûr toujours possible d'utiliser `worker.terminate()` pour l'achever. Cela signifie que le worker se terminera pour toutes les pages qui utilisent ce worker.

Avec un dedicated-worker, lorsque vous fermez la page, vous étiez assuré que le worker disparaissait. Avec les shared-workers, ce n'est plus aussi évident car il se peut qu'une autre page fasse toujours appel à celui-ci. Assurez-vous de bien fermer les shared-workers lorsque vous ne les utilisez plus. Cela vous évitera une consommation de mémoire inutile.

Mais à l'inverse, faites aussi attention à ne pas terminer un shared-worker prématurément s'il est toujours utilisé par une autre page.

Si vous désirez juste interrompre la connexion entre deux threads mais laisser le worker intact pour les autres threads, alors je vous propose d'utiliser la fermeture de port que nous verrons dans la section "Pour en savoir plus".
À présent vous savez manipuler tous les types de workers disponibles en javascript. Vous pouvez dès à présent vous entraîner pour mieux comprendre leur fonctionnement.

Dans la suite de ce tutoriel, nous allons donc mettre en pratique ensemble ces connaissances nouvellement acquises.

TP : Jeu des chiffres et des lettres

Voici venu le moment de vérifier si vous avez bien compris les bases.

Dans ce chapitre, vous allez découvrir un exercice vous permettant de vous familiariser avec les workers.

Présentation de l'exercice

Vous connaissez sans doute le jeu des chiffres et des lettres. Je vous propose de travailler sur ce sujet.

Bien sûr le thème de ce tutoriel correspond aux web-workers. Ainsi je ne vais pas vous demander de tout écrire (bien que si vous en avez envie vous pouvez le faire), mais d'améliorer le code que je vais vous donner.

Le jeu des chiffres

Pour ceux qui ne connaissent pas, le jeu des chiffres dans le jeu "des chiffres et des lettres", consiste à effectuer des opérations (addition, soustraction, multiplication et division) sur 6 nombres afin de trouver un nombre donné (ou s'en approcher le plus possible) dans un temps donné.

Le programme sans worker

Dans cet exercice, je vous propose de partir sur un code existant.

Secret ([cliquez pour afficher](#))

Code : HTML - index.html

```
<!DOCTYPE html>
<html lang="fr">
<head>
  <meta charset="UTF-8" />
  <title>TP : jeu des chiffres</title>
  <style>
    .info{
      position: absolute;
      left: 0;
      right: 0;
      top: 0;
      height: 7em;
    }
    #zonePlaque{
      display: block;
    }
    #jeuCible{ /*le nombre à trouver*/
      display: inline-block;
      width: 4em;
      padding: 0.3em;
      margin: 0.5em;
      letter-spacing: 0.3em;
      font-weight: bold;
      text-align: center;
      font-size: 1.5em;
      color: #0C0;
      text-shadow: 2px 2px 5px #090;
      background-color: #000;
    }
    #jeuTemps{ /*le chronomètre*/
      display: inline-block;
      width: 5em;
      padding: 0.3em;
      margin: 0.5em 0.5em 0.5em 2em;
      text-align: center;
      font-size: 1.5em;
      color: #DC3;
      text-shadow: 2px 2px 2px #00A;
      background-color: #EFE;
      border: 1px solid #000;
    }
    #jeuTemps:after{
      content: " s";
    }
    #zonePlaque > div{ /*les plaques*/
      width: 3em;
      padding: 0.5em;
      margin: 0 0.5em 0 0.5em;
      text-align: center;
      background-color: #000;
      color: #FFF;
      font-weight: bold;
      display: inline-block;
      cursor: pointer;
    }
    #zoneResultat{ /*espace de jeu*/
      position: absolute;
      left: 0;
```

```

    right: 0;
    top: 7em;
    bottom: 0;
    background-color: #CCF;
    display: none;
}
#zoneCalcul{ /*zone d'affichage des opérations*/
    right: 10em;
    padding: 1em;
}
#resultatIA{ /*zone d'affichage de l'IA*/
    position: absolute;
    top: 0.5em;
    bottom: 0;
    right: 0;
    width: 15em;
    background-color: #DDD;
    border: 1px solid #000;
    box-shadow: inset 0 0 0.3em 0.1em #999;
}
#entreeFormule{ /*espace où l'utilisateur entre sa formule*/
    position: absolute;
    display: block;
    bottom: 0.5em;
    left: 0.5em;
}
#jeuDistance{ /*distance obtenue par le joueur*/
    position: absolute;
    display: block;
    bottom: 0.5em;
    right: 17em;
}
#resultatIA > div{ /*distance obtenue par l'IA*/
    position: absolute;
    display: block;
    bottom: 0.5em;
    right: 1em;
}

#zoneCalcul > div, #resultatIA{ /*les opérations*/
    margin: 0.5em 1em 0 0;
    color: #333;
    text-shadow: 1px 1px 0 #111;
    display: block;
    cursor: pointer;
}
#zoneJeu > button{
    float: right;
}

.erreur{
    background-color: #FCC;
}
.compteBon{
    color: #393 !important;
}
</style>
</head>
<body>
<section class="info" id="zoneJeu" style="display:none;">
    <button onclick="initialisation()">Rejouer</button>
    <output id="jeuCible"></output>
    <output id="jeuTemps"></output>
    <section id="zonePlaque"></section>
</section>
<section class="info" id="zoneParametres">
    <label>Nombre de plaques : <input type="number" min="2"
value="6" id="regleNbPlaques"/></label>
    <label>Temps de jeu : <input type="number" min="1" value="45"
id="regleTemps"/> secondes</label>
    <br/>
    <button onclick="initialisation()">Commencer une
partie</button>
</section>
<fieldset id="zoneResultat">
    <legend>Résultat</legend>
    <section id="zoneCalcul"></section>
    <aside id="resultatIA"></aside>
    <input type="text" id="entreeFormule" title="Entrez ici vos
formules de calculs" placeholder="Entrez les calculs ici" />
    <output id="jeuDistance"></output>
</fieldset>

<script src="jeu.js"></script>
</body>
</html>

```

Code : JavaScript - jeu.js

```

/*
Références globales aux éléments HTML fréquemment utilisés
*/

var regleTemps = document.getElementById("regleTemps"); //élément indiquant
le temps total de réflexion
var jeuTemps = document.getElementById("jeuTemps"); //élément indiquant le
temps restant pour jouer
var jeuCible = document.getElementById("jeuCible"); //élément indiquant le
nombre à trouver

var listeNombre = []; //liste des nombres utilisables par l'utilisateur

//paramétrage par défaut
document.getElementById("regleNbPlaques").value=6;
regleTemps.value=45;

// initialisation d'une nouvelle partie
function initialisation(){
  //cache les paramètres de règles
  document.getElementById("zoneParametres").style.display = "none";

  //préparation de la zone de jeu
  document.getElementById("zoneResultat").style.display = "block";
  document.getElementById("zoneJeu").style.display = "block";
  jeuCible.value = "???" ;
  jeuTemps.value = regleTemps.value;

  document.getElementById("zonePlaque").innerHTML = "";
  document.getElementById("resultatIA").innerHTML = "";
  document.getElementById("zoneCalcul").innerHTML = "";

  //initialisation des nombres
  listeNombre = [];
  generateNombre();

  //gestion de l'input servant à entrer un calcul
  var inputFormule = document.getElementById("entreeFormule");
  inputFormule.style.display = "";
  inputFormule.value = "";
  inputFormule.addEventListener("blur", restoreFocus, false);
  inputFormule.addEventListener("keypress", analyseFormule, false);
  inputFormule.addEventListener("blur", analyseFormule, false);
  inputFormule.focus();
}

//gestion du chronometre
var chronometre=(function(){
  var init,timer=-1
  function chrono(){
    var temps = (Date.now() - init)/1000; //temps écoulé depuis le début du je
    jeuTemps.value = Math.round(regleTemps.value - temps);
    if(temps>regleTemps.value){
      //le temps est écoulé
      clearInterval(timer);

      //On retire le formulaire
      var inputFormule = document.getElementById("entreeFormule");
      inputFormule.style.display = "none";
      inputFormule.removeEventListener("blur", restoreFocus, false);
      inputFormule.removeEventListener("keypress", analyseFormule, false);
      inputFormule.removeEventListener("blur", analyseFormule, false);

      //on affiche l'analyse de l'ordinateur
      analyseIA();
    }
  }

  return function(){
    //démarrage du chronomètre
    init = Date.now();
    clearInterval(timer);
    timer = setInterval(chrono,400);
  };
})();

//permet de rechercher une solution et de l'afficher
function analyseIA(){
  //recherche une des meilleures solutions
  var liste = [];
  listeNombre.forEach(function(el){
    if(!el.parent1) liste.push(el.valeur);
  }); //récupération des nombres de départ

```

```

var resultat = chercheSolution(liste, jeuCible.value);
var explication = resultat[1].replace(/\n/g, "<br>");
if(resultat[0]){
  explication += "<div>Compte approchant : " + resultat[0] + "</div>";
}else{
  explication += "<div>Le compte est bon !</div>";
}
document.getElementById("resultatIA").innerHTML = explication;
}

//permet de générer les nombres pour jouer et définit la cible
function generateNombre(){
  var choix =
[1,2,3,4,5,6,7,8,9,10,1,2,3,4,5,6,7,8,9,10,1,2,3,4,5,6,7,8,9,10,25,50,75,100
//plaques possibles
  var nbPlaque = parseInt(document.getElementById("regleNbPlaques").value,10)
  if(listeNombre.length < nbPlaque){
    listeNombre.push(new
Nombre(null,null,null,choix[Math.floor(Math.random()*choix.length)]));
    setTimeout(generateNombre,500);
  }else{
    jeuCible.value = Math.floor(Math.random()*898)+101; //le nombre à trouver
    doit être compris entre 101 et 999
    chronometre(); //on démarre le compte à rebours
  }
}

//permet de redonner le focus à l'input quand il le perd
function restoreFocus(event){
  setTimeout(function(){event.target.focus();},20);
}

//permet d'analyser l'entrée de l'utilisateur
function analyseFormule(event){
  var key = event.keyCode || event.which;
  if(key === undefined || key === 13 || key === 61 || key === 9){ //demande d
valider l'opération

  var operation = this.value.match(/(\d+)\s*([+*\/\-%&X])\s*(\d+)/); //
permet de vérifier que l'opération contient un nombre, un opérateur, et un
nombre
  if(operation){
    var n1 = getNombre(operation[1]),
        n2 = getNombre(operation[3],n1);

    //analyse de l'opérateur utilisé
    switch(operation[2]){
      case "&":
      case "+":
        operation = "+";
        break;
      case " ":
      case "-":
        operation = "-";
        break;
      case "*":
      case "x":
      case "X":
      case "x":
        operation = "x";
        break;
      case "/":
      case "\\":
      case "÷":
        operation = "÷";
        break;
      default:
        operation = null;
    }
  }
  if(operation && n1 && n2){
    //toutes les composantes sont correctes, on peut créer le Nombre
    var n = new Nombre(n1,n2,operation);
    if(n.valeur){ // si n.valeur vaut 0, c'est que l'opération ne respecte pa
les règles
      listeNombre.push(n);
      this.value = "";
      majDistance();
    }
  }else{
    this.className = "erreur";
  }
}
  this.className = ""; // au cas où l'état précédent était en "erreur"
}
}

```

```

//permet de trouver un objet Nombre parmi ceux disponibles qui possède la
valeur recherchée
// valeur : valeur à chercher
// except : continue la recherche si l'objet trouvé est celui indiqué par
except
function getNombre(valeur, except){
  function filtre(el){
    //on ne veut que les objets non utilisés et ayant la bonne valeur
    return !el.UsedBy && el.valeur == valeur && el !== except;
  }
  var liste = listeNombre.filter(filtre); //récupère la liste de tous les
objets correspondant aux critères
  return liste[0]; //seul le premier objet est retourné
}

//met à jour la distance entre les nombres trouvés et la cible
function majDistance(){
  var distance = Infinity;
  var cible = jeuCible.value;
  listeNombre.forEach(function(el) {
    distance = Math.min(distance, Math.abs(el.valeur-cible));
  });
  var jeuDistance = document.getElementById("jeuDistance");
  if(distance){
    jeuDistance.value = "Compte approchant : " + distance;
  }else{
    jeuDistance.value = "Le compte est bon !";
  }
}

//création et affichage d'une plaque
function creationPlaque(nb){
  var plaque = document.createElement("div");
  plaque.textContent = nb;
  plaque.addEventListener("click", ajouteValeur, false);
  document.getElementById("zonePlaque").appendChild(plaque);
  return plaque;
}

//permet d'ajouter la valeur d'une plaque à la formule de calcul
function ajouteValeur(event){
  document.getElementById("entreeFormule").value += event.target.textContent;
}

//Nombre est un objet représentant les nombres manipulés par l'utilisateur
//Il permet de savoir quel nombre a permis de réaliser une opération. Ce qui
facilite le retour en arrière pour supprimer une opération
function Nombre(parent1,parent2,op,init){
  this.parent1 = parent1; //le premier nombre de l'opération
  this.parent2 = parent2; //le deuxième nombre de l'opération
  this.operateur = op; //l'opérateur de l'opération
  this.UsedBy = null; //autre opération qui utilise ce nombre

  if(init){
    this.valeur = init;
    creationPlaque(init);
  }else{
    //réalisation du calcul
    switch(op){
      case "+":
        this.valeur = parent1.valeur + parent2.valeur;
        break;
      case "-":
        this.valeur = parent1.valeur - parent2.valeur;
        break;
      case "x":
        this.valeur = parent1.valeur * parent2.valeur;
        break;
      case "÷":
        this.valeur = parent1.valeur / parent2.valeur;
        break;
    }
    //vérification du calcul
    if(this.valeur < 0 || this.valeur !== Math.round(this.valeur)){
      this.valeur = 0;
      return null;
    }
    this.parent1.utilise(this);
    this.parent2.utilise(this);
    this.createCalcul();
  }
}

//affichage du calcul correspondant à ce nombre
Nombre.prototype.createCalcul = function(){
  this.refCalcul = document.createElement("div");
  this.refCalcul.textContent = this.parent1.valeur + " " + this.operateur + "

```

```

+ this.parent2.valeur + " = " + this.valeur;
if(this.valeur == jeuCible.value){
    this.refCalcul.className = "compteBon";
}else{
    var that = this;
    this.refCalcul.addEventListener("click",function(){that.supprime();},false
)
}
document.getElementById("zoneCalcul").appendChild(this.refCalcul);
};

//définit l'utilisation de ce nombre dans un opération
Nombre.prototype.utilise = function(parent){
    this.usedBy = parent;
};

//définit le fait que ce nombre n'est plus utilisé dans une opération
Nombre.prototype.libre = function(){
    this.usedBy = null;
};

//suppression de ce nombre et donc de l'opération
Nombre.prototype.supprime = function(){
    if(this.usedBy){
        this.usedBy.supprime();
    }
    if(this.parent1){
        this.parent1.libre();
    }
    if(this.parent2){
        this.parent2.libre();
    }
    this.refCalcul.parentNode.removeChild(this.refCalcul);
    listeNombre.splice(listeNombre.indexOf(this),1);
};

//recherche une solution
function chercheSolution(nombres,cible){ //il s'agit d'une fonction récurziv
    var nb1,nb2; //nombres utilisés pour étudier une opération
    var i,j; //index itératifs
    var li = nombres.length; //taille de la liste des nombres parmi lesquels il
    faut chercher le premier nombre de l'opération
    var lj = li - 1; //taille de la liste des nombres moins nb1 parmi lesquels
    le deuxième nombre de l'opération est recherché
    var calcul; //résultat de l'opération en cours
    var rslt; //résultat d'une recherche avec moins de nombres
    var distance = Infinity; //distance de la solution actuelle par rapport à l
    cible
    var solution = ""; //meilleure solution actuelle

    var nombresSansNb1; //liste de nombre sans le premier nombre de l'opération
    (nb1)
    var nombresSansNb2; //liste de nombre sans les nombres de l'opération (nb1
    et nb2)

    for(i=0; i<li && distance; i++){
        nb1 = nombres[i]; //analyse avec ce premier nombre
        nombresSansNb1 = nombres.concat([]); //copie de la liste
        nombresSansNb1.splice(i,1); //on retire le nombre de la liste

        for(j=0; j<lj; j++){
            nb2 = nombresSansNb1[j]; //analyse avec ce deuxième nombre
            nombresSansNb2 = nombresSansNb1.concat([]); //copie de la liste
            nombresSansNb2.splice(j,1); //on retire le nombre de la liste

            //calcul x
            calcul = nb1 * nb2;
            if(Math.abs(cible - calcul)<distance){
                distance = Math.abs(cible - calcul);
                solution = nb1 + " x " + nb2 + " = " + calcul;
                if(!distance) break; //on a trouvé une solution on arrête la boucle
            }
            rslt = chercheSolution(nombresSansNb2.concat([calcul]),cible); // on
            relance la recherche avec les nombres restant + ce résultat
            if(rslt[0]<distance){
                distance = rslt[0];
                solution = nb1 + " x " + nb2 + " = " + calcul + "\n" + rslt[1];
                if(!distance) break; //on a trouvé une solution on arrête la boucle
            }

            //calcul +
            calcul = nb1 + nb2;
            if(Math.abs(cible - calcul)<distance){
                distance = Math.abs(cible - calcul);
                solution = nb1 + " + " + nb2 + " = " + calcul;
                if(!distance) break; //on a trouvé une solution on arrête la boucle
            }
            rslt = chercheSolution(nombresSansNb2.concat([calcul]),cible); // on

```

```

    relance la recherche avec les nombres restant + ce résultat
    if(rslt[0]<distance){
        distance = rslt[0];
        solution = nb1 + " + " + nb2 + " = " + calcul + "\n" + rslt[1];
        if(!distance) break; //on a trouvé une solution on arrête la boucle
    }

    //calcul -
    calcul = nb1 - nb2;
    if(calcul>0){
        if(Math.abs(cible - calcul)<distance){
            distance = Math.abs(cible - calcul);
            solution = nb1 + " - " + nb2 + " = " + calcul;
            if(!distance) break; //on a trouvé une solution on arrête la boucle
        }
        rslt = chercheSolution(nombresSansNb2.concat([calcul]),cible); // on
        relance la recherche avec les nombres restant + ce résultat
        if(rslt[0]<distance){
            distance = rslt[0];
            solution = nb1 + " - " + nb2 + " = " + calcul + "\n" + rslt[1];
            if(!distance) break; //on a trouvé une solution on arrête la boucle
        }
    }

    //calcul ÷
    calcul = nb1 / nb2;
    if(calcul === Math.floor(calcul)){
        if(Math.abs(cible - calcul)<distance){
            distance = Math.abs(cible - calcul);
            solution = nb1 + " ÷ " + nb2 + " = " + calcul;
            if(!distance) break; //on a trouvé une solution on arrête la boucle
        }
        rslt = chercheSolution(nombresSansNb2.concat([calcul]),cible); // on
        relance la recherche avec les nombres restant + ce résultat
        if(rslt[0]<distance){
            distance = rslt[0];
            solution = nb1 + " ÷ " + nb2 + " = " + calcul + "\n" + rslt[1];
            if(!distance) break; //on a trouvé une solution on arrête la boucle
        }
    }
}

return [distance,solution];
}

```

[Tester le jeu sans les workers !](#)

Ce code permet à un utilisateur de jouer au jeu des Chiffres, mais aussi de présenter une solution au problème qui a été posé. L'algorithme pour trouver cette solution est un simple brute-force qui analyse tous les cas possibles.

Des problèmes

En testant ce jeu, vous devriez remarquer un problème de taille : lorsque le temps est écoulé, la page se fige pendant un certain temps.

Effectivement lorsque le temps est écoulé, l'ordinateur se met à chercher une solution. Et dans certains cas (surtout quand le bon compte n'est pas atteignable) cette recherche prend du temps.

Dans certains cas, avec des ordinateurs un peu lents, on peut même voir apparaître la fameuse alerte demandant d'interrompre le script.

Même si c'est fonctionnel, l'utilisateur peut en ressentir une très grande gêne.

À vous de jouer

Le but de l'exercice consiste à améliorer la performance de la recherche de solution afin que l'utilisateur soit le moins gêné possible par cette recherche.

Une façon de résoudre cet exercice pourrait consister à améliorer l'algorithme de recherche. C'est tout à fait possible mais c'est compliqué 🤔

Une solution beaucoup plus simple consiste à réaliser la recherche pendant que le joueur réfléchit à sa solution. Ainsi l'ordinateur dispose du même temps que le joueur pour trouver une solution, ce qui est souvent largement suffisant.

Et là si vous avez bien suivi le cours, une solution technique devrait vous sauter aux yeux : **utiliser un worker !**

Correction

N'oubliez pas qu'il n'y a pas une seule solution. Mais celle que je vais vous montrer peut vous donner des pistes sur la manière

d'intégrer un worker dans un projet.

Identifier les besoins

Tout d'abord il est important de repérer les fonctions qui seront mises dans un worker.

Nous souhaitons paralléliser la recherche de solution, la fonction qui réalise cette action est `chercheSolution`.

Cette fonction peut être appelée indépendamment des autres. Il s'agit donc de la seule fonction à déplacer.

Il faut donc créer un fichier `solution.js` qui contiendra cette fonction.

Instanciation du worker

Si le navigateur ne supporte pas les workers, il est possible de créer un contournement en chargeant dynamiquement le fichier `solution.js`.

Dans le cas où les workers sont supportés, on démarre un nouveau thread.

Code : JavaScript - jeu.js

```
if(window.Worker){
  //le navigateur supporte les workers
  var solutionWorker = new Worker("./solution.js");
  solutionWorker.addEventListener("message", reponseWorker, false);
}else{
  //le navigateur ne supporte pas les Workers, on charge le fichier
  dynamiquement
  (function(){//pour éviter de polluer l'espace globale
    var script = document.createElement("script");
    script.src="./solution.js";
    document.body.appendChild(script);
  })();
}
```

À ce stade, le code doit toujours fonctionner si le navigateur ne supporte pas les workers. Pour essayer, vous pouvez mal orthographier la condition `if(window.Worker)` afin de la rendre fausse et charger le script sans Worker.

La communication avec le Worker

Avant de s'occuper de la communication, on va analyser plusieurs cas de fonctionnement. Par exemple on peut avoir un cas où le jeu se termine avant que la solution ne soit encore trouvée. Dans ce cas, dès que nous avons la réponse il faut l'afficher. Ensuite on a le cas inverse (qui devrait être plus fréquent), où il faudra conserver la réponse jusqu'à la fin du jeu.

Pour pouvoir distinguer ces cas, nous allons ajouter une propriété à notre worker (il s'agit d'un objet comme les autres), qui nous servira à conserver la réponse ou indiquer que la réponse doit être envoyée immédiatement.

Ainsi dans la fonction `analyseIA`, on sépare les différents cas :

Code : JavaScript - jeu.js

```
//permet de rechercher une solution et de l'afficher
function analyseIA(){
  if(solutionWorker){
    //les workers sont utilisés
    if(solutionWorker.resultat){
      //le résultat a déjà été trouvé
      affichageIA(solutionWorker.resultat);
    }else{
      //le résultat n'a pas encore été trouvé
      solutionWorker.resultat = -1;
      document.getElementById("resultatIA").textContent = "recherche en
cours..."; //on avertit l'utilisateur
    }
  }else{
    //Le worker n'est pas utilisé ainsi on lance la recherche de
solutions
    var liste = [];
    listeNombre.forEach(function(el){
      if(!el.parent1) liste.push(el.valeur);
    }); //récupération des nombres de départ
    affichageIA(chercheSolution(liste, jeuCible.value));
  }
}

//affiche le résultat trouvé par l'ordinateur
function affichageIA(resultat){
  var explication = resultat[1].replace(/\n/g, "<br>");
  if(resultat[0]){
    explication += "<div>Compte approchant : " + resultat[0] +
"</div>";
  }else{
    explication += "<div>Le compte est bon !</div>";
  }
}
```

```

    explication += "<div>Le compte est bon :</div> ";
  }
  document.getElementById("resultatIA").innerHTML = explication;
}

```

Recevoir des messages du worker

On peut maintenant s'occuper de la fonction `reponseWorker` qui analysera les réponses du worker.

Code : JavaScript - jeu.js

```

//permet d'analyser la réponse du worker
function reponseWorker(event){
  var reponse = event.data.split("|");
  if(solutionWorker.resultat === -1){
    //il faut afficher le resultat tout de suite
    affichageIA(reponse);
  }else{
    //on garde la réponse au chaud
    solutionWorker.resultat = reponse;
  }
}

```

Démarrer la recherche

Pour ce qui est du démarrage du calcul, contrairement à l'architecture sans worker, il doit se faire au début du jeu. Il faut donc modifier la fonction `generateNombre` pour démarrer le calcul à ce moment.

Code : JavaScript - jeu.js

```

//permet de générer les nombres pour jouer et défini la cible
function generateNombre(){
  var choix =
    [1,2,3,4,5,6,7,8,9,10,1,2,3,4,5,6,7,8,9,10,1,2,3,4,5,6,7,8,9,10,25,50,75,100];
  //plaques possibles
  var nbPlaque = parseInt(document.getElementById("regleNbPlaques").value,10);
  if(listeNombre.length < nbPlaque){
    listeNombre.push(new
    Nombre(null,null,null,choix[Math.floor(Math.random()*choix.length)]));
    setTimeout(generateNombre,500);
  }else{
    jeuCible.value = Math.floor(Math.random()*898)+101; //le nombre à trouver
    doit être compris entre 101 et 999
    if(solutionWorker){
      //on utilise un worker pour trouver la solution
      var liste = "";
      listeNombre.forEach(function(el){liste += el.valeur+",";}); //on prépare
      la liste des nombres
      liste += jeuCible.value; //on ajoute la cible à la fin de la liste
      solutionWorker.postMessage(liste); //on envoie la liste pour démarrer la
      recherche
      solutionWorker.resultat = null; //on réinitialise la propriété resultat
    }
    chronometre(); //on démarre le compte à rebours
  }
}

```

Le worker

Maintenant, il reste à nous occuper de la communication dans le worker. Il faut donc démarrer le calcul en fonction des informations reçues.

Code : JavaScript - solution.js

```

//listener qui permet de contrôler le worker
onmessage = function(event){
  var nombres = event.data.split(","); //on récupère la liste des
  nombres
  var cible = parseInt(nombres.pop(),10); //on récupère la cible
  nombres = nombres.map(function(v){return parseInt(v,10);}); //
  conversion de toutes les valeurs en nombre
  var resultat = chercheSolution(nombres,cible); //on effectue le
  calcul
  postMessage(resultat.join("|")); //on renvoie la solution
};

```



À la ligne 7, on convertit le résultat qui se présente sous forme de tableau, en chaîne de caractères. Dans la partie avancée du tutoriel, nous verrons qu'il est possible d'envoyer autre chose que des chaînes de caractères, et qu'il est possible de se passer de cette conversion.

Cas particulier

Comme il est possible d'exécuter plusieurs codes en même temps, il faut regarder si des cas particuliers ne se présentent pas. Par exemple : Si l'utilisateur demande de recommencer à jouer alors que l'ordinateur est toujours en train de chercher la solution.



Que se passe-t-il ?

Le jeu va recommencer, on aura donc de nouveaux nombres, et ces nombres seront à nouveau envoyés au worker. Or celui-ci est déjà en train de chercher la réponse au jeu précédent (qui a été arrêté). Il ne s'occupera de ces nouveaux nombres que lorsqu'il aura fini avec les premiers.

Ainsi, si le jeu se termine avant qu'il ait fini avec ces derniers, il affichera la solution des premiers nombres qui n'ont rien à voir avec les nombres en cours.

Et même si on se dit qu'il reste une bonne marge, l'utilisateur peut très bien cliquer sur recommencer 20 fois de suite. Et l'ordinateur recherchera une solution pour tous ces jeux.

Une solution consiste à arrêter le worker grâce à `terminate` lorsqu'un nouveau jeu est démarré. Ainsi pour faire simple il suffit de tuer le worker et d'en recréer un autre à chaque fois qu'on initialise une nouvelle partie, même si le worker avait fini son travail.

Comme le worker peut être créé à différents endroits, autant créer une fonction :

Code : JavaScript - jeu.js

```
//création d'un worker
function createWorker() {
  var w = new Worker("./solution.js");
  w.addEventListener("message", reponseWorker, false);
  return w;
}

if(window.Worker) {
  //le navigateur supporte les workers
  var solutionWorker = createWorker();
}
```

Et dans la fonction initialisation, il suffit d'ajouter :

Code : JavaScript - jeu.js

```
if(solutionWorker) {
  //pour éviter de calculer la solution d'une autre partie on tue le
  worker
  solutionWorker.terminate();
  solutionWorker = createWorker();
}
```

Et voilà le jeu est prêt !

Solution avec un worker

Secret (cliquez pour afficher)

Code : HTML - index.html

```
<!DOCTYPE html>
<html lang="fr">
<head>
  <meta charset="UTF-8" />
  <title>TP : jeu des chiffres</title>
</style>
.info{
  position: absolute;
  left: 0;
  right: 0;
  top: 0;
  height: 7em;
}
#zonePlaque{
  display: block;
}
#jeuCible{ /*le nombre à trouver*/
```

```

    display: inline-block;
    width: 4em;
    padding: 0.3em;
    margin: 0.5em;
    letter-spacing: 0.3em;
    font-weight: bold;
    text-align: center;
    font-size: 1.5em;
    color: #0C0;
    text-shadow: 2px 2px 5px #090;
    background-color: #000;
}
#jeuTemps{ /*le chronomètre*/
display: inline-block;
width: 5em;
padding: 0.3em;
margin: 0.5em 0.5em 0.5em 2em;
text-align: center;
font-size: 1.5em;
color: #DC3;
text-shadow: 2px 2px 2px #00A;
background-color: #EFE;
border: 1px solid #000;
}
#jeuTemps:after{
content: " s";
}
#zonePlaque > div{ /*les plaques*/
width: 3em;
padding: 0.5em;
margin: 0 0.5em 0 0.5em;
text-align: center;
background-color: #000;
color: #FFF;
font-weight: bold;
display: inline-block;
cursor: pointer;
}
#zoneResultat{ /*espace de jeu*/
position: absolute;
left: 0;
right: 0;
top: 7em;
bottom: 0;
background-color: #CCF;
display: none;
}
#zoneCalcul{ /*zone d'affichage des opérations*/
right: 10em;
padding: 1em;
}
#resultatIA{ /*zone d'affichage de l'IA*/
position: absolute;
top: 0.5em;
bottom: 0;
right: 0;
width: 15em;
background-color: #DDD;
border: 1px solid #000;
box-shadow: inset 0 0 0.3em 0.1em #999;
}
#entreeFormule{ /*espace où l'utilisateur entre sa formule*/
position: absolute;
display: block;
bottom: 0.5em;
left: 0.5em;
}
#jeuDistance{ /*distance obtenue par le joueur*/
position: absolute;
display: block;
bottom: 0.5em;
right: 17em;
}
#resultatIA > div{ /*distance obtenue par l'IA*/
position: absolute;
display: block;
bottom: 0.5em;
right: 1em;
}

#zoneCalcul > div, #resultatIA{ /*les opérations*/
margin: 0.5em 1em 0 0;
color: #333;
text-shadow: 1px 1px 0 #111;
display: block;
cursor: pointer;
}
#zoneJeu > button{

```

```

        float: right;
    }

    .erreur{
        background-color: #FCC;
    }
    .compteBon{
        color: #393 !important;
    }
</style>
</head>
<body>
    <section class="info" id="zoneJeu" style="display:none;">
        <button onclick="initialisation()">Rejouer</button>
        <output id="jeuCible"></output>
        <output id="jeuTemps"></output>
        <section id="zonePlaque"></section>
    </section>
    <section class="info" id="zoneParametres">
        <label>Nombre de plaques : <input type="number" min="2"
value="6" id="regleNbPlaques"/></label>
        <label>Temps de jeu : <input type="number" min="1" value="45"
id="regleTemps"/> secondes</label>
        <br/>
        <button onclick="initialisation()">Commencer une
partie</button>
    </section>
    <fieldset id="zoneResultat">
        <legend>Résultat</legend>
        <section id="zoneCalcul"></section>
        <aside id="resultatIA"></aside>
        <input type="text" id="entreeFormule" title="Entrez ici vos
formules de calculs" placeholder="Entrez les calculs ici" />
        <output id="jeuDistance"></output>
    </fieldset>

    <script src="jeu.js"></script>
</body>
</html>

```

Code : JavaScript - jeu.js

```

/*
Références globales aux éléments HTML fréquemment utilisés
*/

var regleTemps = document.getElementById("regleTemps"); //élément indiquant
le temps total de réflexion
var jeuTemps = document.getElementById("jeuTemps"); //élément indiquant le
temps restant pour jouer
var jeuCible = document.getElementById("jeuCible"); //élément indiquant le
nombre à trouver

var listeNombre = []; //liste des nombres utilisables par l'utilisateur

//paramétrage par défaut
document.getElementById("regleNbPlaques").value=6;
regleTemps.value=45;

// initialisation d'une nouvelle partie
function initialisation(){
    if(solutionWorker){
        //pour éviter de calculer la solution d'une autre partie
        solutionWorker.terminate();
        solutionWorker = createWorker();
    }

    //cache les paramètres de règles
    document.getElementById("zoneParametres").style.display = "none";

    //préparation de la zone de jeu
    document.getElementById("zoneResultat").style.display = "block";
    document.getElementById("zoneJeu").style.display = "block";
    jeuCible.value = "???" ;
    jeuTemps.value = regleTemps.value;

    document.getElementById("zonePlaque").innerHTML = "";
    document.getElementById("resultatIA").innerHTML = "";
    document.getElementById("zoneCalcul").innerHTML = "";

    //initialisation des nombres
    listeNombre = [];
    generateNombre();
}

```

```

//gestion de l'input servant à entrer un calcul
var inputFormule = document.getElementById("entreeFormule");
inputFormule.style.display = "";
inputFormule.value = "";
inputFormule.addEventListener("blur", restoreFocus, false);
inputFormule.addEventListener("keypress", analyseFormule, false);
inputFormule.addEventListener("blur", analyseFormule, false);
inputFormule.focus();
}

//gestion du chronometre
var chronometre=(function(){
  var init,timer=-1;
  function chrono(){
    var temps = (Date.now() - init)/1000; //temps écoulé depuis le début du jeu
    jeuTemps.value = Math.round(regleTemps.value - temps);
    if(temps>regleTemps.value){
      //le temps est écoulé
      clearInterval(timer);

      //On retire le formulaire
      var inputFormule = document.getElementById("entreeFormule");
      inputFormule.style.display = "none";
      inputFormule.removeEventListener("blur", restoreFocus, false);
      inputFormule.removeEventListener("keypress", analyseFormule, false);
      inputFormule.removeEventListener("blur", analyseFormule, false);

      //on affiche l'analyse de l'ordinateur
      analyseIA();
    }
  }

  return function(){
    //démarrage du chronomètre
    init = Date.now();
    clearInterval(timer);
    timer = setInterval(chrono,400);
  };
})();

//permet de rechercher une solution et de l'afficher
function analyseIA(){
  if(solutionWorker){
    //les workers sont utilisés
    if(solutionWorker.resultat){
      //le résultat a déjà été trouvé
      affichageIA(solutionWorker.resultat);
    }else{
      //le résultat n'a pas encore été trouvé
      solutionWorker.resultat = -1;
      document.getElementById("resultatIA").textContent = "recherche en
cours..."; //on avertit l'utilisateur
    }
  }else{
    //Le worker n'est pas utilisé ainsi on lance la recherche de solutions
    var liste = [];
    listeNombre.forEach(function(el){
      if(!el.parent1) liste.push(el.valeur);
    }); //récupération des nombres de départ
    affichageIA(chercheSolution(liste,jeuCible.value));
  }
}

//affiche le résultat trouvé par l'ordinateur
function affichageIA(resultat){
  var explication = resultat[1].replace(/\n/g, "<br>");
  if(resultat[0]>0){
    explication += "<div>Compte approchant : " + resultat[0] + "</div>";
  }else{
    explication += "<div>Le compte est bon !</div>";
  }
  document.getElementById("resultatIA").innerHTML = explication;
}

//permet de générer les nombres pour jouer et définit la cible
function generateNombre(){
  var choix =
[1,2,3,4,5,6,7,8,9,10,1,2,3,4,5,6,7,8,9,10,1,2,3,4,5,6,7,8,9,10,25,50,75,100
//plaques possibles
  var nbPlaque = parseInt(document.getElementById("regleNbPlaques").value,10)
  if(listeNombre.length < nbPlaque){
    listeNombre.push(new
Nombre(null,null,null,choix[Math.floor(Math.random()*choix.length)]));
    setTimeout(generateNombre,500);
  }else{
    jeuCible.value = Math.floor(Math.random()*898)+101; //le nombre à trouver
  }
}

```

```

doit être compris entre 101 et 999
    if(solutionWorker){
        //on utilise un worker pour trouver la solution
        var liste = "";
        listeNombre.forEach(function(el){liste += el.valeur+",";}); //on prépare
la liste des nombres
        liste += jeuCible.value; //on ajoute la cible à la fin de la liste
        solutionWorker.postMessage(liste); //on envoie la liste pour démarrer la
recherche
        solutionWorker.resultat = null; //on réinitialise la propriété resultat
    }
    chronometre(); //on démarre le compte à rebours
}

//permet de redonner le focus à l'input quand il le perd
function restoreFocus(event){
    setTimeout(function(){event.target.focus();},20);
}

//permet d'analyser l'entrée de l'utilisateur
function analyseFormule(event){
    var key = event.keyCode || event.which;
    if(key === undefined || key === 13 || key === 61 || key === 9){ //demande d
valider l'opération

    var operation = this.value.match(/(\d+)\s*([+\-*/\&xX])\s*(\d+)/); //
permet de vérifier que l'opération contient un nombre, un opérateur, et un
nombre
    if(operation){
        var n1 = getNombre(operation[1]),
            n2 = getNombre(operation[3],n1);

        //analyse de l'opérateur utilisé
        switch(operation[2]){
            case "&":
            case "+":
                operation = "+";
                break;
            case " ":
            case "-":
                operation = "-";
                break;
            case "*":
            case "x":
            case "X":
            case "x":
                operation = "x";
                break;
            case "/":
            case "\\":
            case "÷":
                operation = "÷";
                break;
            default:
                operation = null;
        }
    }
    if(operation && n1 && n2){
        //toutes les composantes sont correctes, on peut créer le Nombre
        var n = new Nombre(n1,n2,operation);
        if(n.valeur){ // si n.valeur vaut 0, c'est que l'opération ne respecte pa
les règles
            listeNombre.push(n);
            this.value = "";
            majDistance();
        }
    }else{
        this.className = "erreur";
    }
}
}
this.className = ""; // au cas où l'état précédent était en "erreur"
}

//permet de trouver un objet Nombre parmi ceux disponibles qui possède la
valeur recherchée
// valeur : valeur à chercher
// except : continue la recherche si l'objet trouvé est celui indiqué par
except
function getNombre(valeur, except){
    function filtre(el){
        //on ne veut que les objets non utilisés et ayant la bonne valeur
        return !el.usedBy && el.valeur == valeur && el !== except;
    }
    var liste = listeNombre.filter(filtre); //récupère la liste de tous les
objets correspondant aux critères
    return liste[0]; //seul le premier objet est retourné

```

```

}

//met à jour la distance entre les nombres trouvés et la cible
function majDistance(){
    var distance = Infinity;
    var cible = jeuCible.value;
    listeNombre.forEach(function(el){
        distance = Math.min(distance,Math.abs(el.valeur-cible));
    });
    var jeuDistance = document.getElementById("jeuDistance");
    if(distance){
        jeuDistance.value = "Compte approchant : " + distance;
    }else{
        jeuDistance.value = "Le compte est bon !";
    }
}

//création et affichage d'une plaque
function creationPlaque(nb){
    var plaque = document.createElement("div");
    plaque.textContent = nb;
    plaque.addEventListener("click",ajouteValeur,false);
    document.getElementById("zonePlaque").appendChild(plaque);
    return plaque;
}

//permet d'ajouter la valeur d'une plaque à la formule de calcul
function ajouteValeur(event){
    document.getElementById("entreeFormule").value += event.target.textContent;
}

//Nombre est un objet représentant les nombres manipulés par l'utilisateur
//Il permet de savoir quel nombre a permis de réaliser une opération. Ce qui
//facilite le retour en arrière pour supprimer une opération
function Nombre(parent1,parent2,op,init){
    this.parent1 = parent1; //le premier nombre de l'opération
    this.parent2 = parent2; //le deuxième nombre de l'opération
    this.operateur = op; //l'opérateur de l'opération
    this.usedBy = null; //autre opération qui utilise ce nombre

    if(init){
        this.valeur = init;
        creationPlaque(init);
    }else{
        //réalisation du calcul
        switch(op){
            case "+":
                this.valeur = parent1.valeur + parent2.valeur;
                break;
            case "-":
                this.valeur = parent1.valeur - parent2.valeur;
                break;
            case "x":
                this.valeur = parent1.valeur * parent2.valeur;
                break;
            case "÷":
                this.valeur = parent1.valeur / parent2.valeur;
                break;
        }
        //vérification du calcul
        if(this.valeur < 0 || this.valeur !== Math.round(this.valeur)){
            this.valeur = 0;
            return null;
        }
        this.parent1.utilise(this);
        this.parent2.utilise(this);
        this.createCalcul();
    }
}

//affichage du calcul correspondant à ce nombre
Nombre.prototype.createCalcul = function(){
    this.refCalcul = document.createElement("div");
    this.refCalcul.textContent = this.parent1.valeur + " " + this.operateur + " " + this.parent2.valeur + " = " + this.valeur;
    if(this.valeur == jeuCible.value){
        this.refCalcul.className = "compteBon";
    }else{
        var that = this;
        this.refCalcul.addEventListener("click",function(){that.supprime();},false);
    }
    document.getElementById("zoneCalcul").appendChild(this.refCalcul);
};

//définit l'utilisation de ce nombre dans un opération
Nombre.prototype.utilise = function(parent){
    this.usedBy = parent;
}

```



```

};

//définit le fait que ce nombre n'est plus utilisé dans une opération
Nombre.prototype.libre = function() {
  this.usedBy = null;
};

//suppression de ce nombre et donc de l'opération
Nombre.prototype.supprime = function() {
  if(this.usedBy) {
    this.usedBy.supprime();
  }
  if(this.parent1) {
    this.parent1.libre();
  }
  if(this.parent2) {
    this.parent2.libre();
  }
  this.refCalcul.parentNode.removeChild(this.refCalcul);
  listeNombre.splice(listeNombre.indexOf(this), 1);
};

//permet d'analyser la réponse du worker
function reponseWorker(event) {
  var reponse = event.data.split("|");
  console.log(reponse[2]/1000);
  if(solutionWorker.resultat === -1) {
    //il faut afficher le resultat tout de suite
    affichageIA(reponse);
  } else {
    //on garde la réponse au chaud
    solutionWorker.resultat = reponse;
  }
}

//création d'un worker
function createWorker() {
  var w = new Worker("./solution.js");
  w.addEventListener("message", reponseWorker, false);
  return w;
}

if(window.Worker) {
  //le navigateur supporte les workers
  var solutionWorker = createWorker();
} else {
  //le navigateur ne supporte pas les Workers, on charge le fichier
  dynamiquement
  (function() { //pour éviter de polluer l'espace globale
    var script = document.createElement("script");
    script.src = "./solution.js";
    document.body.appendChild(script);
  })();
}

```

Code : JavaScript - solution.js

```

//fichier utilisé par le worker

//recherche une solution
function chercheSolution(nombres, cible) { //il s'agit d'une
  fonction récursive
  var nb1, nb2; //nombres utilisés pour étudier une opération
  var i, j; //index itératifs
  var li = nombres.length; //taille de la liste des nombres parmi
  lesquels il faut chercher le premier nombre de l'opération
  var lj = li - 1; //taille de la liste des nombres moins nb1
  parmi lesquels le deuxième nombre de l'opération est recherché
  var calcul; //résultat de l'opération en cours
  var rslt; //résultat d'une recherche avec moins de nombres
  var distance = Infinity; //distance de la solution actuelle par
  rapport à la cible
  var solution = ""; //meilleure solution actuelle

  var nombresSansNb1; //liste de nombre sans le premier nombre de
  l'opération (nb1)
  var nombresSansNb2; //liste de nombre sans les nombres de
  l'opération (nb1 et nb2)

  for(i=0; i<li && distance; i++){
    nb1 = nombres[i]; //analyse avec ce premier nombre
    nombresSansNb1 = nombres.concat([]); //copie de la liste
    nombresSansNb1.splice(i, 1); //on retire le nombre de la liste

```

```

    for(j=0; j<1j; j++){
        nb2 = nombresSansNb1[j]; //analyse avec ce deuxième nombre
        nombresSansNb2 = nombresSansNb1.concat([]); //copie de la
liste
        nombresSansNb2.splice(j,1); //on retire le nombre de la liste

        //calcul x
        calcul = nb1 * nb2;
        if(Math.abs(cible - calcul)<distance){
            distance = Math.abs(cible - calcul);
            solution = nb1 + " x " + nb2 + " = " + calcul;
            if(!distance) break; //on a trouvé une solution on arrête la
boucle
        }
        rslt = chercheSolution(nombresSansNb2.concat([calcul]),cible);
        // on relance la recherche avec les nombres restant + ce résultat
        if(rslt[0]<distance){
            distance = rslt[0];
            solution = nb1 + " x " + nb2 + " = " + calcul + "\n" + rslt[1];
            if(!distance) break; //on a trouvé une solution on arrête la
boucle
        }

        //calcul +
        calcul = nb1 + nb2;
        if(Math.abs(cible - calcul)<distance){
            distance = Math.abs(cible - calcul);
            solution = nb1 + " + " + nb2 + " = " + calcul;
            if(!distance) break; //on a trouvé une solution on arrête la
boucle
        }
        rslt = chercheSolution(nombresSansNb2.concat([calcul]),cible);
        // on relance la recherche avec les nombres restant + ce résultat
        if(rslt[0]<distance){
            distance = rslt[0];
            solution = nb1 + " + " + nb2 + " = " + calcul + "\n" + rslt[1];
            if(!distance) break; //on a trouvé une solution on arrête la
boucle
        }

        //calcul -
        calcul = nb1 - nb2;
        if(calcul>0){
            if(Math.abs(cible - calcul)<distance){
                distance = Math.abs(cible - calcul);
                solution = nb1 + " - " + nb2 + " = " + calcul;
                if(!distance) break; //on a trouvé une solution on arrête la
boucle
            }
            rslt = chercheSolution(nombresSansNb2.concat([calcul]),cible);
            // on relance la recherche avec les nombres restant + ce résultat
            if(rslt[0]<distance){
                distance = rslt[0];
                solution = nb1 + " - " + nb2 + " = " + calcul + "\n" +
rslt[1];
            if(!distance) break; //on a trouvé une solution on arrête la
boucle
            }
        }

        //calcul ÷
        calcul = nb1 / nb2;
        if(calcul === Math.floor(calcul)){
            if(Math.abs(cible - calcul)<distance){
                distance = Math.abs(cible - calcul);
                solution = nb1 + " ÷ " + nb2 + " = " + calcul;
                if(!distance) break; //on a trouvé une solution on arrête la
boucle
            }
            rslt = chercheSolution(nombresSansNb2.concat([calcul]),cible);
            // on relance la recherche avec les nombres restant + ce résultat
            if(rslt[0]<distance){
                distance = rslt[0];
                solution = nb1 + " ÷ " + nb2 + " = " + calcul + "\n" +
rslt[1];
            if(!distance) break; //on a trouvé une solution on arrête la
boucle
            }
        }

    }

    }

    return [distance,solution];
}

//listener qui permet de contrôler le worker
onmessage = function(event){

```

```
var d = Date.now();
var nombres = event.data.split(","); //on récupère la liste des
nombres
var cible = parseInt(nombres.pop(),10); //on récupère la cible
nombres = nombres.map(function(v){return parseInt(v,10)}); //
conversion de toutes les valeurs en nombre
var resultat = chercheSolution(nombres,cible); //on effectue le
calcul
resultat.push(Date.now() - d);
postMessage(resultat.join("|")); //on renvoie la solution
};
```

Essayer le jeu avec un worker !

Et voilà, vous avez vu comment intégrer un worker dans un projet existant 😊

Maintenant que vous avez assimilé les bases, vous allez pouvoir étudier les workers plus en profondeur.

Partie 2 : Pour tout savoir sur les web-workers

Maintenant que vous connaissez les bases, nous allons pouvoir analyser tous les détails qui feront de vous un maître des workers.

Tout est communication

Maintenant que vous savez manipuler les workers, nous allons regarder tous les détails qui tourment autour. Vous allez bientôt tout savoir. Absolument tout ! ou presque...

Comme les workers n'ont qu'un accès restreint au monde extérieur, et afin qu'ils puissent fournir le résultat de leur dur labeur, il faut bien qu'ils transmettent leurs informations correctement aux autres threads. La communication avec les autres threads est donc un élément essentiel du fonctionnement des workers.

Cela ne vous surprendra donc pas si je clame que **tout est communication** ; et que je dédie tout un chapitre à ce sujet.

Nous allons étudier ici comment créer des canaux de communications et comment les manipuler. Nous verrons ensuite des détails qui ont été passés sous silence pendant la prise en main. Et enfin on verra une méthode pour arrêter une communication sans arrêter le worker.

Le canal de communication

Un canal de communication est ce qui permet de gérer la communication entre les éléments qui communiquent. Autant dire que, dans notre cas, c'est très important.

MessageChannel

Depuis le début je vous parle des web-workers, mais je vous ai fait manipuler, sans vous le dire, un deuxième objet issu des API HTML5 : les **message-channels** (ou, en français, les canaux de communications).

Avec les shared-workers, je vous ai parlé des ports. Un port est toujours attaché à un autre port. Ces deux objets sont liés. Si on envoie un message à un de ces objets, l'autre le reçoit. Ainsi lorsque ces deux ports appartiennent à deux threads distincts, il suffit qu'un thread envoie un message à travers son port pour que l'autre thread puisse le récupérer. Il s'agit de la base de la communication entre les threads.

Et bien **le canal de communication correspond au lien entre ces deux ports**.

Lorsque vous créez un worker, l'interpréteur se charge de créer en même temps deux ports liés et les place chacun dans un thread différent. Ce qui nous permet de communiquer facilement entre ces threads. Mais grâce à l'objet `MessageChannel`, vous pouvez créer explicitement deux ports liés entre eux, et vous pourrez en faire ce que vous voulez.

L'objet `MessageChannel` créé possède deux objets : **port1** et **port2**. Comme leurs noms l'indiquent il s'agit de ports. Et, vous l'aurez deviné, ces deux ports sont liés.

Code : JavaScript

```
var channel = new MessageChannel(); //on a créé un canal de
communication entre channel.port1 et channel.port2

channel.port1.onmessage=function(e) { //on écoute sur le port1
  alert(e.data);
};

channel.port2.postMessage("Bonjour"); //on envoie un message dans
le port2
```

Essayer !

Vous pouvez remarquer que le message du port2 a bien été reçu par le port1.



C'est bien tout ça, mais c'est un peu parler à soi-même. C'est rarement très utile ...
Ne pourrait-on pas, au moins, envoyer ce port à un autre thread ?

Transférer des objets

Et bien oui on va transférer un port à un autre thread. Pour cela on va utiliser le deuxième argument de `postMessage`. Cet argument permet d'envoyer une liste d'objets transférables. Pour tout vous dire, actuellement, il n'existe qu'un seul type d'objet transférable : les ports (ou plus exactement les `MessagePort`).

Donc l'envoi est simple :

Code : JavaScript

```
//création du canal de communication
var channel = new MessageChannel();
channel.port1.onmessage = function(e) {
  alert(e.data);
};
```

```
//création du worker
var w = new Worker("worker.js");
w.onmessage = function(e) {
  alert("Je ne veux pas recevoir de message par ce canal");
};

//transfert du port2 vers le thread
w.postMessage("Voici le port2", [channel.port2]);
```

Il faut bien retenir que le transfert est une liste, il ne faut donc pas oublier les []. Cela signifie qu'il est possible de transférer plusieurs ports en même temps !

C'est bien, on a pu envoyer un port. Ça serait bien de le récupérer maintenant ! Pour cela on va lire l'attribut **ports** de l'événement onmessage. Et comme on a envoyé une liste d'objets, nous allons récupérer son premier élément avec [0]. Cela ne vous rappelle rien ?

C'est exactement de cette manière qu'on a récupéré le port dans le shared-worker lors de sa connexion !

Voici donc le code de notre worker :

Code : JavaScript

```
self.onmessage = function(e) {
  var port = e.ports[0];
  port.postMessage("Hello canal");
}
```

Essayer !

Maintenant si vous testez ces deux codes, vous devriez obtenir le message "Hello canal" qui a été envoyé par le canal que vous avez créé et non pas par celui du worker !



Avez-vous essayé de faire `channel.port2.postMessage("test")` ; après la ligne 14 dans le premier code ?

Vous obtenez un message d'erreur 🙄

Alors qu'il fonctionne très bien si vous le placez avant la ligne 14 ! Que s'est-il passé ?

C'est parce que votre port a été transféré, il n'est donc plus disponible dans votre premier thread. C'est pourquoi depuis le début de cette section, je vous parle de transfert et d'objet transférable.

En réalité, pour éviter le *cross-site scripting* (exécution de script provenant d'un autre site) et pour rester dans le modèle de séparation des données. Le port a été copié, le lien a été transféré à la copie, puis l'objet original a été supprimé. Ainsi si vous ajoutez des attributs ou définissez des listeners, ils ne seront pas transmis.



N'oubliez pas qu'on écoute sur des ports. Donc si vous utilisez `addEventListener`, vous devez d'abord utiliser `start()` pour recevoir la communication.

C'est bien beau tout ça, mais communiquer avec le worker, on pouvait déjà le faire !

Oui mais parfois il est pratique d'avoir plusieurs points d'entrée. Mais sinon, avec les shared-workers, nous avons plusieurs interlocuteurs en même temps. Grâce au `MessageChannel`, nous pouvons désormais faire communiquer deux pages ensemble sans interlocuteur intermédiaire !

main.js	workers.js
<p>Code : JavaScript</p> <pre>//création d'une zone d'échange var input = document.createElement("input"); input.disabled = true; input.onchange = function() { port.postMessage(this.value); this.value = ""; }; document.body.appendChild(input); //création d'une zone d'affichage var output = document.createElement("output"); output.value = "En attente d'un interlocuteur"; document.body.appendChild(output); var port; //création du worker var w = new</pre>	<p>Code : JavaScript</p> <pre>var port = null; //gestion des connexions onconnect = function(e) { var port2 = e.ports[0]; if(!port) { //première connexion port = port2; } else { //un autre thread est déjà connecté var ch = new MessageChannel();</pre>

<pre>SharedWorker("worker.js"); w.port.onmessage = function(e) { if(e.ports && e.ports.length) { input.disabled = false; output.value = ""; port = e.ports[0]; port.onmessage = function(e) { output.value = e.data; }; w.terminate(); } };</pre>	<pre>messageChannel(); port.postMessage("", [ch.port1]); port2.postMessage("", [ch.port2]); port = null; } };</pre>
Essayer !	

Avec ces codes nous pouvons créer un mini-chat entre deux pages. Vous pouvez remarquer qu'à la ligne 31 on arrête le worker. Donc une fois le port reçu, le worker est tué. Et pourtant la communication fonctionne toujours. Car le canal de communication vit désormais entre les deux pages et non plus via le worker. Malheureusement comme le javascript fonctionne côté client (le navigateur), il n'est pas possible de réaliser un chat entre différents utilisateurs de cette manière. 😞

? Tous les ports sont-ils transférables ?

Non ! Vous ne pouvez pas transférer le port source ou le port de destination dans son propre canal. Et dans une liste de ports vous ne devez pas mettre un port en double. Sinon la diffusion ne se fera pas.

Mieux communiquer En savoir plus sur celui qui nous parle

Quelques détails qui peuvent parfois s'avérer utiles. Lorsqu'un listener de type `onmessage` se déclenche, un événement est créé. À partir de cet événement, on obtient de nombreuses informations :

- `event.data` : On l'a vu plusieurs fois, il permet de récupérer les données.
- `event.ports` : On vient juste de le voir, il permet d'obtenir la liste des objets transférés.
- `event.target` : Il s'agit du port ayant reçu le message. Donc si vous voulez répondre, il suffit d'utiliser cet attribut pour répondre au bon interlocuteur. Dans le cadre des `dedicated-workers`, il est équivalent à `self`. Vous pouvez aussi trouver les attributs `currentTarget`, `srcElement` et parfois `source` qui sont équivalents à celui-ci (mais ils sont plus imprévisibles en fonction des navigateurs).
- `event.origin` : Ce champ nous renseigne sur l'identité de celui qui a envoyé le message. Il s'agit de l'URL de la page appelante. Il n'est pas forcément très utile dans le cadre des web-workers, mais peut s'avérer utile pour vérifier l'origine de l'interlocuteur (et éviter de parler avec des inconnus).
- `event.type` : Il vaut "message" c'est le type de l'événement 😊.

Communiquer avec des objets

Vous avez peut-être remarqué que tous les messages qui ont été envoyés n'étaient que du texte. À un moment vous voudrez forcément transmettre un objet.

Le JSON

La première idée qu'on peut avoir est d'utiliser le JSON. JSON permet de convertir un objet en une chaîne de caractères. Il suffit de transformer l'objet en chaîne de caractères avant de l'envoyer, puis de le retransformer en objet à l'arrivée.

Code : JavaScript

```
var message=JSON.stringify(obj); //on sérialise l'objet
var nvObj=JSON.parse(message); //on décode le message pour
reconstruire l'objet
```

Le JSON n'est pas l'objectif de ce tutoriel, je ne vais donc pas détailler son fonctionnement davantage.

Toutefois le JSON a ses limites.

Premièrement tous les types des objets sont perdus. Ils deviennent de simples objets, même pour les objets "fondamentaux" tels que les objets `Date`.

Ensuite le JSON n'aime pas du tout les références circulaires.

Code : JavaScript

```
var obj1 = {};  
var obj2 = {};  
  
obj1.frere = obj2;  
obj2.frere = obj1;  
  
var pere = {enfant:[obj1,obj2]};
```

Dans ce code, il n'est pas possible de "stringifier" l'objet `pere` avec JSON, à cause de la référence circulaire entre `obj1` et `obj2`.

Envoyer des objets javascript

Heureusement il est tout à fait possible d'envoyer des objets javascript à travers nos canaux de communications. Cependant il faut faire attention, car tous les objets ne sont pas acceptés. Seuls ceux qui sont clonables peuvent être envoyés (les structured clone). Les objets clonables correspondent aux variables primitives (nombres, chaînes de caractères, booléens...), les Arrays et les objets contenant des attributs clonable.

Ainsi l'objet `{msg:"message", nb:10, liste:[{txt:"texte"}, false]}` sera accepté mais l'objet `{a:10, fct:function() {return this.a;}}` sera refusé car les fonctions ne sont pas clonables.



Contrairement aux objets transférés, un objet cloné reste disponible dans le thread d'origine.

Le premier avantage de cloner des objets est que le type des objets "fondamentaux" est conservé.

Par exemple si une des références est un objet de type `Array`, `Date`, `RegExp`, `ImageData`, `File`, `FileList`, ou `Blob`, leurs références et leur type seront conservés.

Par contre si vous clonez un objet que vous avez créé à partir d'un constructeur "maison", il perd son type et ses prototypes.

Le deuxième gros avantage est qu'il est tout à fait possible de cloner des objets ayant des références circulaires. Ainsi dans l'exemple précédent, il est tout à fait possible de cloner l'objet `pere`.



Si l'objet que l'on souhaite envoyer contient des objets de type `Error` ou `Function` alors une erreur de type `DataCloneError` se produira et aucune donnée ne sera transmise.
De même, s'il y a une référence à un nœud DOM (attaché ou non au document).

Certaines propriétés ne sont pas clonées comme la propriété `lastIndex` d'un objet de type `RegExp`.

Les getters et les setters (ainsi que toutes les propriétés associées aux metadata) ne sont pas clonés.

Le prototype (et toutes ses références) n'est pas cloné non plus.

Terminer une communication

`close()` : arrêter la communication

Nous avons vu l'utilisation de `terminate()` pour arrêter un worker. Cependant, parfois, on désirerait simplement couper la connexion sans arrêter le worker. Par exemple dans les cas d'utilisation des `shared-workers` ou de `message-channel`.

Je vous présente maintenant une autre méthode, plus douce, pour arrêter une communication : `close()`. `close` permet de fermer la communication sur laquelle elle est appliquée. Le worker n'est pas arrêté, seul le canal de communication désigné est interrompu.

Il faut l'appliquer directement sur le port visé : `port.close()` ;

Quand `close` est invoqué. Il n'est plus possible de redémarrer cette connexion. Le canal de communication est détruit. Si vous voulez redémarrer une communication, il faudra recréer un canal de communication avec `MessageChannel`.

onclose : quand la communication s'arrête

Il peut être particulièrement utile de savoir quand la communication est interrompue. Pour cela vous pouvez utiliser le listener `onclose`. Cependant cette méthode n'appartient pas aux spécifications ! Les navigateurs ne sont donc pas tenus de l'implémenter.

Actuellement, seul Firefox semble l'avoir implémenté.

Il va donc falloir patienter avant de pouvoir vraiment l'utiliser 😞

Maintenant que la communication n'a plus de secret pour vous, nous allons pouvoir continuer dans le prochain chapitre la découverte des workers et voir toutes les possibilités qu'ils nous offrent 😊

Manipuler les workers

Dans ce chapitre, nous verrons les derniers détails concernant l'utilisation des workers.

Nous allons regarder comment appréhender les erreurs qui se déclenchent dans les workers. Nous allons découvrir comment obtenir des informations sur l'environnement d'exécution. Et nous allons aussi voir comment triturer un peu plus les workers.

La gestion des erreurs

Il est possible que vos codes ne soient pas parfaits (pas la peine de se cacher, ça arrive même aux meilleurs). Il peut donc arriver que le worker plante, il vous sera alors utile de le savoir afin de le redémarrer ou plus simplement pour déboguer.

onerror : détecter une erreur

Pour se tenir au courant de la moindre erreur, il suffit d'utiliser le listener **onerror**. Et pour récupérer le message d'erreur, vous pouvez lire la propriété **message** de l'objet événement. Vous pourrez aussi avoir besoin des propriétés suivantes :

- **filename**, qui indique le fichier qui a produit l'erreur ;
- **lineno**, qui nous renseigne sur ligne où l'erreur s'est produite.

Le listener peut être placé sur le worker, ou dans le worker. Si une erreur se produit dans le worker, ces deux listeners seront déclenchés.



Même si vous pouvez le trouver dans quelques navigateurs, il n'existe pas officiellement de fonction `postError`. Car le `onerror` est censé détecter un problème majeur dans le code.

Toutefois si vous tenez à générer une erreur, vous pouvez utiliser **throw** pour envoyer une erreur, et s'il n'y a pas de **try{ } catch ()** l'erreur sera alors propagée et même jusqu'au thread parent via le `onerror`.

Vous pouvez aussi créer un événement représentant une erreur et le propager.

Bien sûr, il est tout à fait possible d'utiliser `addEventListener`.

main.js	worker.js
<p>Code : JavaScript</p> <pre>//création du worker var w = new Worker("worker.js"); //réception des messages w.addEventListener("message",function(event){ alert("Le worker a répondu : " + event.data); if(event.data.substr(0,7) == "Bonjour"){ var nom = prompt("Quel est ton nom ?"); w.postMessage("L'utilisateur s'appelle : "+ nom); } },false); //gestion des erreurs w.addEventListener("error",function(event){ alert("Il a planté ! Son excuse est :\n"+ event.message+ "\n\nfichier : "+event.filename+ "\n\nligne : "+event.lineno); },false); w.postMessage("Bonjour");</pre>	<p>Code : JavaScript</p> <pre>//gestion des messages function dialogue(event){ if(event.data == "Bonjour"){ throw "J'aime pas dire bonjour"; } } addEventListener("message",dialogue,false); //gestion des erreurs en interne addEventListener("error",function(e){ postMessage("J'ai une erreur : "+ e.message); },false);</pre>
Essayer !	



Il n'y a aucune raison que les erreurs soient propagées à travers un canal de communication. On ne peut donc pas les observer à travers les ports de `MessageChannel`, ni à travers les ports des `shared-workers`.

Comportement des navigateurs

Il est à noter que les navigateurs ne sont pas encore parfaits sur ce point. Par exemple Opera n'écouterait pas toutes les erreurs à l'intérieur du worker si vous utilisez `addEventListener("error", f, false)` ;

Avec les `shared-Workers`, aucun navigateur actuel ne semble récupérer les erreurs sur le worker. Heureusement ils les capturent quand même à l'intérieur du worker.

Débugger un worker

Débugger les workers s'avère plus difficile que du code 'normal' pour la simple raison qu'en tant qu'utilisateur on ne se trouve pas dans le même thread. Avec la séparation des données, il n'est pas simple de savoir ce qu'il se passe à l'intérieur du worker. Et donc récupérer les erreurs devient un allié très utile.

Tous les navigateurs ont maintenant une console intégrée, qui permet d'afficher des informations très utiles. Cependant à l'intérieur des workers cette console n'est pas accessible. Certains navigateurs ont donc contourné les problèmes. Avec Opera, la console est partagée avec celle du thread principal. Ceci est particulièrement pratique mais contredit le principe de zones distinctes. Avec Webkit, il est possible d'ouvrir un espace de développement dans le worker grâce au "worker inspection". Cet espace de développement offre une console dédiée au worker et donc d'envoyer des messages dedans. Mais aussi de poser des breakpoints dans le worker afin de l'analyser pas à pas. Il y a de fortes chances qu'à terme tous les navigateurs proposeront une fonctionnalité similaire.

Vous pouvez aussi trouver sur internet des codes qui permettent de créer une référence à la console dans le worker en redirigeant son contenu dans celle du thread principal. Par exemple le script de David Flanagan pour créer une console partagée grâce au MessageChannel.

Sinon vous pouvez vous contenter d'envoyer via le `postMessage` vos informations de debug en les encapsulant dans un objet.

Par exemple :

main.js	worker.js
<p>Code : JavaScript</p> <pre>var w = new Worker("worker.js"); w.addEventListener("message", function(event) { if(typeof event.data === "object" && event.data.cmd === "debug") { //il s'agit d'un message de debug console.debug(event.data.msg); } }, false);</pre>	<p>Code : JavaScript</p> <pre>function debug(obj) { postMessage({cmd: "debug", msg: obj}); } /* maintenant, à n'importe quel endroit où l'on souhaite analyser la valeur d'un objet il suffit de faire : debug(objet); et sa valeur s'affichera dans la console. ATTENTION: l'objet doit être clonable */</pre>

Connaître ses origines

Afin d'obtenir des informations sur le contexte d'exécution, deux objets sont disponibles dans l'espace global du worker : **location** et **navigator**.

location

L'objet location permet d'obtenir des informations sur l'appel du fichier d'origine qui fait fonctionner le script. Toutes les propriétés sont en read-only, il n'est donc pas possible de les modifier comme on peut le faire avec `window.location`.

Pour indiquer un exemple de ce que retournent les propriétés, on va considérer que notre worker a été appelé de cette manière `new Worker('http://www.siteduzero.com:80/tutoriel/javascript/toto.js?tata=tutu#titi');`

- **href** : Il s'agit de l'url complète (`http://www.siteduzero.com:80/tutoriel/javascript/toto.js?tata=tutu#titi`)
- **protocol** : Il s'agit du protocole utilisé (`http`)
- **host** : Il s'agit du serveur mais aussi du port utilisé (`http://www.siteduzero.com:80`)
- **hostname** : Cette fois, il ne s'agit que du serveur (`http://www.siteduzero.com`)
- **port** : Et là, il ne s'agit que du port (`80`)
- **pathname** : Il s'agit du chemin relatif depuis le serveur (`/tutoriel/javascript/toto.js`)
- **search** : Il s'agit de la partie derrière le ? (`?tata=tutu`)
- **hash** : Il s'agit de la partie derrière le # (`#titi`)



Ces propriétés sont proches de celle de `window.location`, toutefois `self.location` ne comporte aucune méthode. Ceci peut facilement s'expliquer par le fait que les propriétés sont ici en lecture seule. Et même `toString()` ne retournera pas l'url comme on pourrait s'y attendre (il faut alors utiliser `self.location.href`).



Avec les shared-workers, pour qu'un code soit partagé il doit être appelé avec la même location. Ainsi si le hash ou le search est différent, les codes seront distincts et ne seront pas partagés.

navigator

L'objet `navigator` permet d'obtenir des informations sur le navigateur dans lequel le script s'exécute.

- **appName** : Il s'agit du nom officiel du navigateur.
- **appVersion** : Il s'agit de la version du navigateur (sous forme de chaîne de caractères).
- **platform** : Il s'agit du nom de la plate-forme dans laquelle le navigateur fonctionne (Windows, Linux,...).
- **userAgent** : Il s'agit de la chaîne d'identification complète du navigateur.
- **online** : Permet d'indiquer si le navigateur travaille "en-ligne" ou non.

Ajouter un script dynamiquement : `importScripts`

Dans un worker, il est bien sûr possible d'utiliser `XMLHttpRequest` pour lire le contenu d'un fichier grâce à l'AJAX.

Toutefois si vous désirez charger un script dynamiquement, cela se complique fortement. En effet, vous ne pouvez pas créer de balise script et exécuter un code externe de cette manière. Pour pallier ce problème, il existe une nouvelle fonction disponible uniquement dans les workers : **`importScripts`**.

Cette fonction permet de charger de manière synchrone un code situé dans un fichier.

Supposons que vous ayez écrit une API nommée `calcul.js`.

Code : JavaScript

```
function addition(a, b){  
  return a + b;  
}
```

Dans votre worker, vous pouvez écrire :

Code : JavaScript

```
importScripts("calcul.js");  
var resultat = addition(1, 2); // resultat vaut 3
```

À la ligne 1, l'interpréteur va charger et exécuter le code situé dans le fichier `calcul.js`. Puis, une fois fini, il va exécuter la deuxième ligne.

Il est aussi possible d'inclure directement plusieurs fichiers d'un coup :

Code : JavaScript

```
importScripts("calcul1.js", "calcul2.js", "calcul3.js",  
"calcul4.js");
```

L'ordre de chargement est du premier au dernier (de gauche à droite).



Le contexte d'exécution des scripts importés est toujours l'espace global du worker (donc `self`).

Ce point est très important ! Examinons quelques exemples pour mieux comprendre sa signification.

Si on a dans le fichier `toto.js` :

Code : JavaScript

```
a = b + c;
```

Et dans le worker :

Code : JavaScript

```
var a = 10,  
    b = 20,  
    c = 30;
```

```
importScripts("toto.js");
var resultat = a; //vaut 50
```

Effectivement la variable `a` a été modifiée dans le fichier importé.
Ceci nous confirme le caractère synchrone de l'import.

Intégrons maintenant cet import dans une fonction (c'est-à-dire qu'il est appelé dans un contexte d'exécution qui est différent de l'espace global). Notre worker ressemble à ceci :

Code : JavaScript

```
var a = 10,
    b = 20,
    c = 30;

function test() {
  var a = 1,
      b = 2,
      c = 3;
  importScripts("toto.js");
  var resultat1 = a; //vaut 1
  var resultat2 = self.a; //vaut 50
}

test();
```

Bien que `importScripts` ait été appelé dans la fonction, son environnement de travail est l'espace global. Ce sont donc les variables `a`, `b` et `c` de `self` qui ont été modifiées et non pas les variables locales de la fonction `test`.

Il ne s'agit donc pas d'une "copie" de code à l'endroit où il est appelé (comme pour l'`include` en PHP) mais bien d'une exécution de code.



Si le même fichier est importé dans plusieurs workers, il s'exécutera de manière indépendante dans chaque worker. Ses actions ne seront pas partagées entre les workers, mais seront bien effectives dans l'environnement de chaque worker.

Enchaîner les workers

Et oui ! Il est tout à fait possible d'enchaîner les workers !

Ainsi dans un worker, il est possible de créer un ou plusieurs nouveau(x) worker(s).

main.js	worker1.js	worker2.js
<p>Code : JavaScript</p> <pre>// initialisation du worker var w = new Worker("worker1.js"); w.addEventListener("message", function(event){ alert("j'ai reçu le message suivant :\n" +event.data); },false); // on envoie un message w.postMessage("Bonjour");</pre>	<p>Code : JavaScript</p> <pre>if(self.Worker){ //on crée un nouveau worker var w=new Worker("worker2.js"); w.addEventListener("message", function(event){ /* transmission du message au parent (en le modifiant) */ postMessage('worker2 a ce '+ 'message pour toi : '+ event.data + ''); }, false); /* gestion de la communication avec le parent */ addEventListener("message", function(event){ /* transmission du message au worker (sans changement) */ w.postMessage(event.data); }, false); } else{ /*le navigateur ne gère</pre>	<p>Code : JavaScript</p> <pre>addEventListener("message", function(event){ postMessage("Merci de m'écrire"); }, false);</pre>

```

pas
les workers dans un worker*/
postMessage("Votre
navigateur "+
"ne supporte pas les
Workers"+
" dans les Workers ☹");
}

```

Essayer !

Ce code transmet un message au premier worker qui le transmet au deuxième. Celui-ci répond, le premier worker le reçoit et le modifie puis le retransmet au script d'origine.

Vous aurez compris par vous même que `worker2.js` ne peut pas communiquer directement avec `main.js`, puisqu'il ne le connaît pas. Il est obligé de passer par son parent pour lui adresser un message.

Il est bien sûr possible de créer plusieurs workers par thread et donc de gérer toute une ribambelle de scripts de cette manière



Bien que cela soit possible, il n'est pas vraiment recommandé d'enchaîner trop de workers. Principalement à cause de leur consommation de ressources. Comme nous le verrons dans le prochain chapitre.

Lorsque vous créez un worker, vous lui adressez une url relative au script en cours. Ainsi dans la page HTML, l'origine se trouve au niveau de la page HTML.

Dans un worker, l'origine se trouve au niveau du fichier s'exécutant dans le worker.

Imaginons que vous ayez deux répertoires nommés *html* et *scripts* qui contiennent respectivement vos fichiers html et vos fichiers javascript. Ces répertoires sont situés tous deux dans le même répertoire parent.

Dans un fichier `index.html`, on appelle donc un script de cette manière :

Code : HTML

```
<script src="../../scripts/main.js"></script>
```

Dans ce fichier javascript on pourra donc créer un worker en l'appelant de cette manière :

Code : JavaScript

```
var w = new Worker("../scripts/worker1.js");
```

Dans ce worker, si l'on veut créer un autre worker, on devra faire :

Code : JavaScript

```
var w = new Worker("worker2.js");
```

Faites donc bien attention à l'origine du script en cours d'exécution lorsque vous créez un worker.

Créer un worker inline

Dans certains cas, il peut être pratique de créer un worker dans le même code que celui qui va l'appeler.

Je tiens à préciser que cette méthode n'est à réserver que pour quelques cas très limités. En fait, je vous conseille même de l'éviter tant que vous le pouvez. Mais comme je vous ai promis de tout vous montrer sur les workers, je vais quand même vous le montrer.

Pour créer un worker il nous faut une URL.

Nous allons donc créer un **Blob** et créer une url vers son contenu.

Création du BLOB

Pour créer un BLOB, il suffit d'utiliser l'objet **BlobBuilder** (sous certains navigateurs, il faut le préfixer, cela donnera donc `WebKitBlobBuilder` ou `MozBlobBuilder`).

Code : JavaScript

```
var blob = new BlobBuilder();
```

Il faut ensuite le remplir, pour cela on utilise la fonction **append**. Par exemple :

Code : JavaScript

```
blob.append("onmessage = function(e){ postMessage('je réponds au
```

```
message : ' + e.data); });");
```

Bien sûr, vu que c'est une chaîne de caractères, vous pouvez la modifier comme vous le voulez avec du contenu issu du DOM ou d'ailleurs. Mais une fois créé, le contenu du worker ne sera plus accessible.

Création de l'url

Pour créer une url vers le blob, on va utiliser la fonction **createObjectURL** de l'objet URL (sous certains navigateurs, il faut le préfixer, ce qui donnera `webkitURL` ou `MozURL`).

Code : JavaScript

```
var blobUrl = URL.createObjectURL(blob.getBlob());
```

Création du worker

Ce qui nous donne au final :

Code : JavaScript

```
var blob = new BlobBuilder();
blob.append("onmessage = function(e){ postMessage('je réponds au
message : ' + e.data); });");
var blobUrl = URL.createObjectURL(blob.getBlob());

var worker = new Worker(blobUrl);
worker.onmessage = function(e) {
    alert(e.data);
};

worker.postMessage("Bonjour");
```

[Essayer !](#)

Maintenant, vous savez tout sur les workers. Il ne vous reste plus qu'à les utiliser et à profiter des nouvelles possibilités qui s'offrent à vous.

Dans le prochain chapitre je vous propose de voir quand il est sage de les utiliser ou non.

Quand utiliser des web-workers ?

Vous savez maintenant presque tout sur les workers, vous devez alors être tenté de les utiliser partout. Mais les workers n'ont pas que des avantages 😞

Dans ce cours, nous allons voir quand il est judicieux d'utiliser ou non les workers.
Dans quels contextes se révèlent-ils utiles ? C'est ce que nous allons voir tout de suite.

De par leur apparition récente, il est assez difficile de faire une liste exacte de bonnes pratiques. Cette partie se contentera donc de donner des idées sur ce qui paraît cohérent et des pistes pour en faire un bon usage.

Quand ne pas utiliser les workers ?

Vous venez d'apprendre à utiliser les workers et, je le comprends, vous voulez maintenant les utiliser. Toutefois je vais vous arrêter 😞 Certes, comme nous allons le voir dans la section suivante, les workers ont un intérêt. Mais il faut garder en tête que la programmation concurrente apporte aussi une complexité à votre code.

Nous allons donc lister les cas où les workers sont inutiles.

- Ne les utilisez pas quand vous n'en avez pas besoin !
Si vous n'avez pas besoin de réaliser plusieurs tâches en même temps, alors vous n'avez pas besoin de la parallélisation et donc des workers.

Jusqu'à présent, vous aviez dû écrire de nombreux codes javascript sans ressentir le besoin d'utiliser la programmation parallèle. En avez-vous vraiment besoin maintenant ?
- Si vous avez besoin d'un résultat avant de continuer l'exécution du script, l'utilisation des workers n'est pas forcément la plus judicieuse car il vous faudra gérer le caractère asynchrone de la réponse.
- Attention aux performances ! Contrairement aux idées reçues, les workers n'améliorent que rarement la performance de votre code. Et il faut aussi tenir compte du temps de transfert des messages.
Les workers n'amélioreront pas la performance de votre code d'un coup de baguette magique. Il est possible de l'améliorer grâce aux workers mais il faut savoir les utiliser correctement. Sinon, dans la plupart des cas, elles seront dégradées.
Nous reviendrons sur la performance, dans la dernière section de ce chapitre.

Quand utiliser les workers ?

Ne désespérez pas !

Les workers sont utiles. Il faut juste les utiliser à bon escient 😊.

D'une manière générale, puisqu'ils n'ont pas accès à l'interface, les workers se prêtent bien aux calculs.

Voici une petite liste où les workers apportent un réel intérêt :

- Quand vous cherchez à réaliser plusieurs actions en même temps. Par exemple vous cherchez à réaliser un calcul mais vous souhaitez que l'utilisateur puisse toujours accéder à d'autres fonctionnalités.
- Quand vous désirez afficher l'état d'avancement d'une tâche sans trop impacter son temps de calcul.
Attention à ne pas vous tromper de sens. Dans la majorité des cas, c'est le thread actif (le calcul) qui doit envoyer son état au thread d'affichage. Libre ensuite à celui-ci d'exploiter ou non cette information. Si vous communiquez dans l'autre sens, le thread d'affichage demande au thread de calcul où il en est (comme cela est souvent effectué avec d'autres langages pour ce genre de réalisation), le thread n'y répondra pas avant d'avoir fini. Effectivement, lorsque le thread reçoit le message, il est mis en attente. Il ne sera lu que lorsque le calcul aura rendu la main, et il y a donc de fortes chances pour qu'il ne rende la main qu'une fois le calcul terminé.
- Dans certains cas, il est possible d'améliorer les performances de votre calcul.
Attention, les cas réels, où l'optimisation est possible, sont plutôt rares. Pour que cela puisse bien marcher, il est absolument nécessaire que le calcul soit distribuable entre plusieurs tâches, et que chaque tâche soit indépendante. Mais il est possible d'obtenir un résultat sans avoir à attendre la fin d'une autre tâche 😊
- Anticiper un résultat. Vous pouvez calculer des informations qui seront utilisées plus tard. Par exemple, dans un jeu, vous pouvez calculer la solution en arrière-plan pendant que l'utilisateur cherche par lui-même. Ainsi l'utilisateur ne subit aucun ralentissement pendant que la solution est recherchée par votre programme. Et la solution lui sera proposée instantanément quand il la demandera. Il s'agit d'une amélioration de performance facile à mettre en place s'il est possible d'anticiper une demande (qu'elle provienne de l'utilisateur ou d'une fonction qui sera utilisée plus tard).
- Dans un modèle en couches, les workers peuvent permettre de s'assurer qu'un thread sera dédié à l'affichage et un autre (ou plusieurs) au moteur. Chaque couche pouvant être représentée par un thread.
- Les shared-workers offrent la possibilité de surveiller la présence d'un utilisateur sur plusieurs pages de votre site ou même plusieurs fois la même page. Ils permettent ainsi de mettre à jour toutes ses pages actives en fonction de ses actions sur les autres. Ou de le prévenir qu'il a déjà ouvert cette page dans un autre onglet ou une autre fenêtre.
On peut aussi imaginer, par ce biais, d'utiliser les onglets du navigateur pour utiliser son site au lieu de créer ses propres onglets (ce cas ne devrait être envisagé que dans le cas d'une application afin de ne pas gêner l'utilisateur dans sa manière de naviguer)

Je ne peux pas lister tous les cas de figure, car cela appelle à votre imagination et à votre innovation. Mais sachez que de nouvelles possibilités vous sont maintenant ouvertes.

La performance

Pour terminer ce chapitre, je vais garder quelques mots pour la performance. Si vous avez l'intention d'utiliser les workers sans vous soucier des problèmes de performances alors pas de soucis. Mais si la performance est un de vos sujets de préoccupation, alors vous devrez lire ces lignes.

La performance est un sujet délicat. Tout d'abord parce que les meilleures améliorations viendront d'abord de l'architecture de votre programme. Ensuite parce que les résultats peuvent varier fortement d'un navigateur à l'autre. Mais en ce qui concerne les workers, il y a des règles immuables.

Tout d'abord il faut bien garder en tête qu'un worker coûte cher. Non pas en argent ! Mais en ressources, surtout en mémoire. Le navigateur doit gérer un espace global supplémentaire ...

Et leur initialisation prend un certain temps (qui n'existe pas quand on ne les utilise pas).

Généralement pour démarrer un calcul, il faudra lui envoyer un message. Et si ce message est volumineux (afin de paramétrer correctement le calcul), la transmission de ce message peut prendre beaucoup de temps. D'autant plus, comme on l'a vu au chapitre précédent, que les messages sont copiés et non transférés.

Le temps nécessaire à l'émission et/ou à la réception ne sont pas à négliger. Surtout si vous recherchez l'optimisation d'un calcul avec des workers.

Il faut donc les utiliser avec parcimonie. Mais surtout un worker est fait pour vivre ! Si vous les utilisez pour améliorer vos performances alors faites-les vivre le plus longtemps possible. Si vous les utilisez pour des petits calculs alors le temps d'initialisation aura fait perdre tout l'intérêt de la parallélisation.

Donc plus le worker vit (et est utilisé) longtemps, plus le rendement sera bon.

Il ne faut pas oublier aussi que la plupart des navigateurs implémentent les workers en thread et non en processus. Du coup, si plusieurs workers travaillent en même temps, ils ne vont pas forcément bénéficier du multicœur de la machine ; ils vont se partager le même temps de fonctionnement alloué au processus. Ils vont donc se ralentir les uns les autres.

Avant d'utiliser un worker pour des raisons de performance, il est donc nécessaire de bien mesurer le coût de fonctionnement d'un worker. Si vous estimez que malgré ces inconvénients vous gagnerez encore en performance, alors n'hésitez plus : foncez !



Cette petite section est loin d'être exhaustive. D'autant plus qu'il est encore bien trop tôt pour donner des règles précises. Mais cela vous donne déjà un bon aperçu où l'utilisation de workers apporte un intérêt ou au contraire sont déconseillés.





Le support des navigateurs

Le support des navigateurs a toujours été un problème en javascript. Cela reste d'autant plus vrai avec l'HTML5, où les fonctionnalités sont ajoutées au fur et à mesure et chaque navigateur a un rythme de développement différent (et des priorités différentes).

Nous allons voir ici ce qu'il en ait à propos des web-workers et de ce qui tourne autour.

Tour d'horizon

Voici un récapitulatif du support des web-workers par les principaux navigateurs :

	Chrome	Firefox	Internet Explorer	Opera	Safari
Worker	3+	3.5+	10+	10.60+	4+
SharedWorker	5+			10.60+	5+
MessageChannel	5+		10+	10.60+	5+
Inline	8+	6+	???		5.1+

Les **dedicated-workers** peuvent être utilisés sans soucis sur tous les navigateurs récents et même des plus anciens (mis à part Internet Explorer). La seule inquiétude reste donc si vous ciblez un usage professionnel où les utilisateurs restent souvent avec de vieilles versions d'Internet Explorer.

Le support des **shared-workers** est plus délicat, puisque les deux principaux navigateurs (en terme de parts de marché) ne les supportent pas. Cela signifie qu'il n'est actuellement pas possible de construire un site ou une application recherchant une large diffusion où les shared-workers sont absolument nécessaires. Ils doivent donc rester optionnels et n'apporter qu'un confort ou une aide de plus.



En plus de ces supports majeurs, il y a également quelques difficultés de compatibilité pour certaines fonctionnalités. Je vous ai présenté les fonctionnalités telles qu'elles devraient fonctionner d'après les spécifications. Nous allons voir maintenant en détail les problèmes des principaux navigateurs.

Dans un but d'impartialité, les navigateurs sont présentés par ordre alphabétique. Ces informations ont été testées sur la dernière version stable disponible du navigateur au moment de l'écriture de ce tutoriel. Elles sont donc sujettes à modification.

Chrome / Safari (Webkit)

Je présente Chrome et Safari en même temps, vu qu'ils utilisent le même moteur, leur comportement est très proche.

1. En local : Pour des raisons de sécurité, les workers ne sont pas appelés dans les pages situées en local sur votre ordinateur.
Il est nécessaire de mettre les pages en ligne pour faire fonctionner les workers. Ou alors il faut démarrer le navigateur avec une option permettant de lire les fichiers locaux.
`Chrome --disable-web-security` ou `Chrome --allow-file-access-from-files` (en fonction de la plateforme ou des versions utilisées)
Le mot clef Chrome doit être remplacé par le nom de votre navigateur (`chromium-browser` pour Chromium, `Safari` pour Safari)
2. Chaînage de workers : Il n'est pas possible d'insérer un nouveau Worker à l'intérieur d'un autre Worker.
Il s'agit d'un choix technique de la part de ses développeurs pour des soucis de performance. Espérons qu'un jour cette fonctionnalité soit également disponible pour ces navigateurs.
3. Le nombre de workers : Pour des raisons de performance, il n'est possible de créer qu'un maximum de 50 workers environ par page, puis la page crashe. Raison de plus pour bien terminer vos workers.
4. Lorsqu'un listener de type "message" se déclenche, l'attribut `origin` vaut "" (chaîne vide). Ce qui est assez dommage pour identifier celui qui nous envoie le message.
5. Il n'est pas possible de fermer un port avec `close()`.
6. Il n'est pas possible de terminer un shared-worker avec `terminate()`. Ce bug est extrêmement contraignant en terme de performance. J'espère qu'il sera corrigé rapidement.
7. Lorsqu'une erreur survient dans un shared-worker, les threads parents ne sont pas prévenus (`worker.onerror` ne se déclenche pas).
8. Lors d'une consommation excessive de mémoire, le worker fait crasher la page. Mis à part l'interruption de tout le code de la page, et de l'inaccessibilité de celle-ci, ce problème n'est pas si grave, car cela permet de protéger l'utilisateur contre un usage abusif du worker.

Firefox

1. Les shared-workers : Les shared-workers ne sont absolument pas supportés.
2. Les MessageChannel : La création d'un canal de communication n'est absolument pas supportée.
3. Lorsqu'un listener de type "message" se déclenche, l'attribut `origin` vaut "" (chaîne vide). Ce qui est assez dommage

pour identifier celui qui nous envoie le message.

4. Lors d'une consommation excessive de mémoire, le worker ne continue plus sa tâche et ne répond plus au thread principal (même au `terminate()`). C'est un comportement extrêmement gênant dans des situations critiques.

Internet Explorer



À l'heure où j'écris ces lignes, Internet Explorer 10 n'est pas encore disponible et je n'ai donc pas pu le tester. Il n'est donc pas possible actuellement de connaître le détail de son support.

1. Les shared-workers : Les shared-workers ne sont absolument pas supportés (selon Microsoft).

Opera

1. Après un `importScripts`, un deuxième `importScripts` avec le ou les mêmes fichiers ne fonctionne pas.
2. Dans un worker, `addEventListener("error", f, false)` ; ne fonctionne pas sauf si `self.onerror` est défini.
3. Avec `self.onerror` (ou avec `addEventListener`), l'événement créé n'est pas un objet mais une chaîne de caractères correspondant au message d'erreur.
4. Lorsqu'une erreur survient dans un shared-worker, les threads parents ne sont pas prévenus (`worker.onerror` ne se déclenche pas).
5. Il n'est pas possible de créer un worker inline (la création d'un Blob et la création d'une URL à partir d'un objet ne fonctionnent pas)
6. Le nombre de workers : Pour des raisons de performance, il n'est possible de créer qu'un maximum de 16 workers par page ou de 128 workers par session. Ensuite toute tentative de créer un nouveau worker générera une erreur de type `QUOTA_EXCEEDED_ERR`. Raison de plus pour bien terminer vos workers. Ces limites sont configurables par l'utilisateur (dans le `opera:config`).

Vous voilà prêt à affronter les différences entre les navigateurs. Mais n'oubliez pas que ces différences peuvent changer à chaque sortie d'une nouvelle version d'un navigateur.

TP : jeu du Gomoku

Nous allons finir ce tutoriel par quelques exercices afin de consolider votre maîtrise des web-workers.

Dans l'exercice principal, nous allons améliorer le confort de l'utilisateur d'un jeu grâce aux web-workers. Vous trouverez aussi à la fin de ce chapitre d'autres exercices mais qui ne seront pas commentés : il n'y aura que l'énoncé et une solution possible.

Les solutions peuvent bien sûr être améliorées, elles n'ont été écrites que pour répondre à l'exercice et rester le plus compréhensible possible.

Présentation de l'exercice

À partir d'un jeu de gomoku, l'ordinateur (une IA) répond aux coups d'un joueur.

Le sujet de cet exercice consiste à rendre la page disponible pendant que l'IA du jeu fonctionne. Le but sera aussi d'informer l'utilisateur de l'avancée de la recherche de l'IA.

Quelques explications

Le gomoku est un jeu de stratégie où le but est d'aligner 5 pions de sa couleur. Un pion joué ne peut plus être bougé. Chaque joueur joue à tour de rôle, les noirs commencent. Si le plateau est rempli et qu'aucun joueur n'a réussi à gagner, la partie est déclarée nulle.

Cela ressemble au morpion, mais sur un plateau plus grand et où il faut aligner plus de pions.

À vous de jouer

N'oubliez pas qu'il n'existe pas une seule solution, et que chacun peut avoir une conception différente. Seul le résultat est important.

Je rappelle que le but est de permettre à l'utilisateur une interface fluide dans laquelle il peut continuer à agir pendant que l'IA cherche une solution. La performance n'est pas le sujet de ce TP.

Le jeu sans worker

Voici les codes de bases d'un jeu de Gomoku.

Secret ([cliquez pour afficher](#))

Le HTML :

Code : HTML

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8"/>
<title>TP : jeu du Morpion</title>
<style>
table{
background-color:#FFCC66;
}
td{
border: 1px solid black;
border-radius: 25px;
width: 25px;
height:25px;
cursor:pointer;
}
td.empty{
background-color:inherit;
}
td.noir{
background-color:#000000;
box-shadow: inset 4px 3px 20px -6px #146717;
}
td.blanc{
background-color:#EEEEFF;
box-shadow: inset 4px 3px 20px -6px #7f8f8f;
}
fieldset{
display: inline-block;
}
fieldset label{
display: block;
}
</style>
</head>
<body>
<script src="./grille.js"></script>
<script src="./IA.js"></script>
</body>
</html>
```

Le javascript permettant de gérer l'affichage de la grille et la gestion du jeu :

Code : JavaScript - grille.js

```
var nx = 7; //nombre de cellules en largeur
var ny = 7; //nombre de cellules en hauteur
var nbAligne = 5; //nombre de jetons à aligner pour gagner
var couleurTour = 1; //couleur dont c'est le tour
var continueJeu = false; //permet d'indiquer si le jeu est arrêté
ou non

var iaProfondeurMax = 4; //indique la profondeur de recherche de
l'IA
var iaNoir = false; //indique si le joueur noir est une IA
var iaBlanc = true; //indique si le joueur blanc est une IA

var grille = []; //grille du jeu
var iaWorker; // worker gérant l'IA (si le navigateur supportent
les workers)

var elemTable; //élément contenant les éléments d'affichage du jeu
var elemIA; //élément indiquant que l'ordinateur réfléchit
var progressIA; //élément permettant d'indiquer où en est
l'ordinateur

//Affichage des éléments pour le paramétrage
function affichageDOM(){
  //Règles
  var fieldset = document.createElement("fieldset");
  var legend = document.createElement("legend");
  legend.textContent = "Règles";
  fieldset.appendChild(legend);

  //NX
  var label = document.createElement("label");
  label.textContent = "Largeur :";
  var inputNX = document.createElement("input");
  inputNX.type="number";
  inputNX.min=1;
  inputNX.value=nx;
  label.appendChild(inputNX);
  fieldset.appendChild(label);

  //NY
  label = document.createElement("label");
  label.textContent = "Hauteur :";
  var inputNY = document.createElement("input");
  inputNY.type="number";
  inputNY.min=1;
  inputNY.value=ny;
  label.appendChild(inputNY);
  fieldset.appendChild(label);

  //aligne
  label = document.createElement("label");
  label.textContent = "Nombre de jetons à aligner pour gagner :";
  var inputAlign = document.createElement("input");
  inputAlign.type="number";
  inputAlign.min=1;
  inputAlign.value=nbAligne;
  label.appendChild(inputAlign);
  fieldset.appendChild(label);

  document.body.appendChild(fieldset);

  //Pour l'IA
  fieldset = document.createElement("fieldset");
  legend = document.createElement("legend");
  legend.textContent = "configuration de l'IA";
  fieldset.appendChild(legend);

  //IA noir?
  label = document.createElement("label");
  label.textContent = "Le joueur noir est un ordinateur :";
  input = document.createElement("input");
  input.type="checkbox";
  input.checked=false;
  input.onchange=function(){
    iaNoir=this.checked;
    iaToPlay(); //on vérifie si c'est au tour de l'IA de jouer
  };
  label.appendChild(input);
  fieldset.appendChild(label);

  //IA blanc?
  label = document.createElement("label");
  label.textContent = "Le joueur blanc est un ordinateur :";
  input = document.createElement("input");
```

```

input.type="checkbox";
input.checked=true;
input.onchange=function(){
  iaBlanc=this.checked;
  iaToPlay(); //on vérifie si c'est au tour de l'IA de jouer
};
label.appendChild(input);
fieldset.appendChild(label);

//Profondeur
label = document.createElement("label");
label.textContent = "Profondeur de recherche :";
input = document.createElement("input");
input.type="number";
input.min=1;
input.value=iaProfondeurMax;
input.onchange=function(){iaProfondeurMax=parseInt(this.value,10);}
;
label.appendChild(input);
fieldset.appendChild(label);

document.body.appendChild(fieldset);

//bouton permettant de lancer la partie
var btnStart = document.createElement("button");
btnStart.textContent = "Commencer";
btnStart.onclick=function(){
  nx = parseInt(inputNX.value,10);
  ny = parseInt(inputNY.value,10);
  nbAligne = parseInt(inputAlign.value,10);
  init();
}
document.body.appendChild(btnStart);

//Indicateur que l'ordinateur réfléchit
elementIA = document.createElement("div");
elementIA.textContent = "L'ordinateur est en train de réfléchir...";
elementIA.style.visibility = "hidden";
document.body.appendChild(elementIA);

document.body.appendChild(document.createElement("hr"));
}

window.addEventListener("load",affichageDOM,false);

//Initialisation d'une partie
function init(){

  //initialisation de la grille
  for(var x=0;x<nx;x++){
    grille[x] = [];
    for(var y=0;y<ny;y++){
      grille[x][y] = 0;
    }
  }

  //suppression de l'élément HTML de la grille précédente
  if(elemTable){
    document.body.removeChild(elemTable);
  }

  //affichage de la grille de jeu
  elemTable = document.createElement("table");
  var row,cel;
  for(y=0;y<ny;y++){
    row = elemTable.insertRow(-1);
    for(x=0;x<nx;x++){
      cel = row.insertCell(-1);
      cel.id = "grille"+x+" "+y;
      cel.onclick=setClick(x,y);
      switch(grille[x][y]){
        case 1:
          cel.className = "noir";
          break;
        case 2:
          cel.className = "blanc";
          break;
        case 0:
        default:
          cel.className = "empty";
      }
    }
  }
  document.body.appendChild(elemTable);
  couleurTour = 1;
  continueJeu = true;

```

```

    iaToPlay(); //on vérifie si c'est au tour de l'IA de jouer
};

//permet de changer la couleur lors d'un coup
function changeCouleur(x,y) {
    grille[x][y]=couleurTour;
    var elem = document.getElementById("grille"+x+"_"+y);
    if(elem){
        elem.className=couleurTour===1?"noir":"blanc";
    }
}

//permet de jouer un coup en x,y
function joue(x,y) {
    if(!continueJeu) return false;
    if(grille[x][y]) return false;
    var rslt;
    changeCouleur(x,y);
    couleurTour = couleurTour%2+1;
    if(rslt=verifVainqueur(x,y)) {
        continueJeu = false;
        alert((rslt===1?"Noirs":"Blancs")+" vainqueurs");
    }

    if(!verifNbLibre()) {
        continueJeu = false;
        alert("Parie nulle : égalité");
    }

    //est-ce que le prochain coup doit être joué par l'IA ?
    iaToPlay();
}

//est-ce que le prochain coup doit être joué par l'IA ?
function iaToPlay() {
    if(!continueJeu) return false;
    if((couleurTour === 1 && iaNoir) || (couleurTour === 2 &&
iaBlanc)) {
        continueJeu = false; //pour empêcher un humain de jouer
        elementIA.style.visibility = "visible";
        setTimeout(function() {
            var rslt = iaJoue(grille,couleurTour);
            continueJeu = true;
            elementIA.style.visibility = "hidden";
            joue(rslt[0],rslt[1]);
        },10); //au cas où deux ordi jouent ensemble et pour voir le coup
        pendant que l'IA réfléchit
    }
}

//permet de créer une fonction listener sur un élément x,y
function setClick(x,y) {
    return function() {
        joue(x,y);
    };
}

//permet de vérifier s'il reste des coups jouables
function verifNbLibre() {
    var nbLibre=0;
    for(var x=0;x<nx;x++) {
        for(var y=0;y<ny;y++) {
            if(grille[x][y]==0) {
                nbLibre++;
            }
        }
    }
    return nbLibre;
}

//permet de vérifier s'il y a un vainqueur (en ne regardant que le
dernier coup joué)
function verifVainqueur(x,y,vGrille) {
    vGrille = vGrille || grille;
    var col = vGrille[x][y]; //couleur du jeton qui vient d'être joué
    var alignH = 1; //nombre de jetons alignés horizontalement
    var alignV = 1; //nombre de jetons alignés verticalement
    var alignD1 = 1; //nombre de jetons alignés diagonalement NO-SE
    var alignD2 = 1; //nombre de jetons alignés diagonalement SO-NE
    var xt,yt;

    //vérification horizontale
    xt=x-1;
    yt=y;
    while(xt>=0 && vGrille[xt][yt]==col) {
        xt--;
        alignH++;
    }
}

```

```

xt=x+1;
yt=y;
while(xt<nx && vGrille[xt][yt]===col){
  xt++;
  alignH++;
}

//vérification verticale
xt=x;
yt=y-1;
while(yt>=0 && vGrille[xt][yt]===col){
  yt--;
  alignV++;
}
xt=x;
yt=y+1;
while(yt<ny && vGrille[xt][yt]===col){
  yt++;
  alignV++;
}

//vérification diagonale NO-SE
xt=x-1;
yt=y-1;
while(xt>=0 && yt>=0 && vGrille[xt][yt]===col){
  xt--;
  yt--;
  alignD1++;
}
xt=x+1;
yt=y+1;
while(xt<nx && yt<ny && vGrille[xt][yt]===col){
  xt++;
  yt++;
  alignD1++;
}

//vérification diagonale SO-NE
xt=x-1;
yt=y+1;
while(xt>=0 && yt<ny && vGrille[xt][yt]===col){
  xt--;
  yt++;
  alignD2++;
}
xt=x+1;
yt=y-1;
while(xt<nx && yt>=0 && vGrille[xt][yt]===col){
  xt++;
  yt--;
  alignD2++;
}

//parmis tous ces résultats on regarde s'il y en a un qui dépasse
le nombre nécessaire pour gagner
if(Math.max(alignH,alignV,alignD1,alignD2)>=nbAligne){
  return col;
}else{
  return 0;
}
}

```

Le javascript permettant de gérer l'Intelligence Artificielle :

Code : JavaScript - IA.js

```

//demande à l'IA de jouer
function iaJoue(grilleOrig,couleur){
  var grille = copieGrille(grilleOrig);
  return iaAlphaBeta(grille, couleur, 0, -Infinity, Infinity);
}

//fonction gérant l'algorithme minimax et l'élagage alpha-beta
function iaAlphaBeta(grille, couleur, profondeur, alpha, beta){
  if(profondeur === iaProfondeurMax){
    //on a atteint la limite de profondeur de calcul on retourne
    donc une estimation de la position actuelle
    if(couleur === 1){
      return iaEstimation(grille);
    }else{
      return -iaEstimation(grille);
    }
  }else{
    var meilleur = -Infinity; //estimation du meilleur coup actuel
    var estim; //estimation de la valeur d'un coup
    var coup=null; //meilleur coup actuel

```

```

    var couleurOpp = couleur%2+1; //optimisation pour calculer la
    couleur adverse

    //on va essayer toutes les combinaisons possibles
    for(var x=0;x<nx;x++){
        for(var y=0;y<ny;y++){
            if(grille[x][y]) continue; //case déjà occupée

            if(!coup){coup=[x,y];} //pour proposer au moins un coup

            grille[x][y]=couleur; //on va essayer avec ce coup
            //vérifie si le coup est gagnant
            if(estim=verifVainqueur(x,y,grille)){
                grille[x][y]=0; //restauration de la grille
                if(!profondeur){
                    return [x,y];
                }else{
                    return Infinity;
                }
            }
            estim = -iaAlphaBeta(grille, couleurOpp, profondeur+1, -beta,
            -alpha); //on calcule la valeur de ce coup

            if(estim > meilleur){
                //on vient de trouver un meilleur coup
                meilleur = estim;
                if(meilleur > alpha){
                    alpha = meilleur;
                    coup = [x,y];
                    if(alpha >= beta){
                        /*ce coup est mieux que le meilleur des coups qui aurait pu
                        être joué si on avait joué un autre
                        coup. Cela signifie que jouer le coup qui a amené cette position
                        n'est pas bon. Il est inutile
                        de continuer à estimer les autres possibilités de cette position
                        (principe de l'élagage alpha-beta). */
                        grille[x][y]=0; //restauration de la grille
                        if(!profondeur){
                            return coup;
                        }else{
                            return meilleur;
                        }
                    }
                }
            }
            grille[x][y]=0; //restauration de la grille
        }
    }
    if(!profondeur){
        return coup;
    }else{
        if(coup) return meilleur;
        else return 0; //si coup n'a jamais été défini c'est qu'il n'y
        a plus de possibilité de jeu. C'est partie nulle.
    }
}

//permet d'estimer la position
function iaEstimation(grille){
    var estimation = 0; //estimation globale de la position

    for(var x=0;x<nx;x++){
        for(var y=0;y<ny;y++){
            if(!grille[x][y]) continue;
            //estimation de la valeur de ce jeton et ajout au calcul
            d'estimation global
            switch(grille[x][y]){
                case 1:
                    estimation += iaAnalyse(grille,x,y);
                    break;
                case 2:
                    estimation -= iaAnalyse(grille,x,y);
                    break;
            }
        }
    }
    return estimation;
}

//permet de calculer le nombre de "libertés" pour la case donnée
function iaAnalyse(grille,x,y){
    var couleur = grille[x][y];
    var estimation = 0; //estimation pour toutes les directions
    var compteur = 0; //compte le nombre de possibilités pour une
    direction
    var centre = 0; //regarde si le jeton a de l'espace de chaque

```

```

côté
var bonus = 0; //point bonus liée aux jetons alliés dans cette
même direction
var i,j; //pour les coordonnées temporaires
var pass=false; //permet de voir si on a passé la case étudiée
var pLiberte = 1; //pondération sur le nombre de liberté
var pBonus = 1; //pondération Bonus
var pCentre = 2; //pondération pour l'espace situé de chaque côt
é

//recherche horizontale
for(i=0;i<nx;i++){
  if(i==x){
    centre = compteur++;
    pass=true;
    continue;
  }
  switch(grille[i][y]){
    case 0: //case vide
      compteur++;
      break;
    case couleur: //jeton allié
      compteur++;
      bonus++;
      break;
    default: //jeton adverse
      if(pass){
        i=nx; //il n'y aura plus de liberté supplémentaire, on
arrête la recherche ici
      }else{
        //on réinitialise la recherche
        compteur = 0;
        bonus = 0;
      }
  }
}
if(compteur>=nbAligne){
  //il est possible de gagner dans cette direction
  estimation += compteur*pLiberte + bonus*pBonus +
(1-Math.abs(centre/(compteur-1)-0.5))*compteur*pCentre;
}

//recherche verticale
compteur=0;
bonus=0;
pass=false;
for(j=0;j<ny;j++){
  if(j==y){
    centre=compteur++;
    pass=true;
    continue;
  }
  switch(grille[x][j]){
    case 0: //case vide
      compteur++;
      break;
    case couleur: //jeton allié
      compteur++;
      bonus++;
      break;
    default: //jeton adverse
      if(pass){
        j=ny; //il n'y aura plus de liberté supplémentaire, on
arrête la recherche ici
      }else{
        //on réinitialise la recherche
        compteur = 0;
        bonus = 0;
      }
  }
}
if(compteur>=nbAligne){
  //il est possible de gagner dans cette direction
  estimation += compteur*pLiberte + bonus*pBonus +
(1-Math.abs(centre/(compteur-1)-0.5))*compteur*pCentre;
}

//recherche diagonale (NO-SE)
compteur=0;
bonus=0;
i=x;
j=y;
while(i-->0 && j-->0){
  switch(grille[i][j]){
    case 0: //case vide
      compteur++;
      break;
    case couleur: //jeton allié

```



```

        compteur++;
        bonus++;
        break;
    default: //jeton adverse, on arrête de rechercher
        i=0;
    }
}
centre=compteur++;
i=x;
j=y;
while(++i<nx && ++j<ny){
    switch(grille[i][j]){
        case 0: //case vide
            compteur++;
            break;
        case couleur: //jeton allié
            compteur++;
            bonus++;
            break;
        default: //jeton adverse, on arrête de rechercher
            i=nx;
    }
}
if(compteur>=nbAligne){
    //il est possible de gagner dans cette direction
    estimation += compteur*pLiberte + bonus*pBonus +
    (1-Math.abs(centre/(compteur-1)-0.5))*compteur*pCentre;
}

//recherche diagonale (NE-SO)
compteur=0;
bonus=0;
i=x;
j=y;
while(i-->0 && ++j<ny){
    switch(grille[i][j]){
        case 0: //case vide
            compteur++;
            break;
        case couleur: //jeton allié
            compteur++;
            bonus++;
            break;
        default: //jeton adverse, on arrête de rechercher
            i=0;
    }
}
centre=compteur++;
i=x;
j=y;
while(++i<nx && j-->0){
    switch(grille[i][j]){
        case 0: //case vide
            compteur++;
            break;
        case couleur: //jeton allié
            compteur++;
            bonus++;
            break;
        default: //jeton adverse, on arrête de rechercher
            i=nx;
    }
}
if(compteur>=nbAligne){
    //il est possible de gagner dans cette direction
    estimation += compteur*pLiberte + bonus*pBonus +
    (1-Math.abs(centre/(compteur-1)-0.5))*compteur*pCentre;
}

return estimation;
}

//permet de copier une grille, cela permet d'éviter de modifier
//par inadvertance la grille de jeu originale
function copieGrille(grille){
    var nvGrille=[];
    for(var x=0;x<nx;x++){
        nvGrille[x]=grille[x].concat([]); //effectue une copie de la
        //liste
    }
    return nvGrille;
}

```

Dans cette IA, l'algorithme principal est un Minimax avec un élagage alpha-beta. Il s'agit d'une recherche en profondeur d'abord. Même si vous ne comprenez pas comment il fonctionne, ce qu'il faut retenir c'est que la fonction principale est `iaAlphaBeta` ; il s'agit d'une fonction récursive (elle se rappelle elle-même en fonction de la profondeur recherchée).

Dans un algorithme Minimax, l'estimation d'une position est très importante. Dans ce code, l'estimation est gérée par les fonctions `iaEstimation` et `iaAnalyse`. L'estimation est basée sur le nombre de pions alliés et le nombre de cases vides autour de chaque pion. Cette estimation est très coûteuse, et il existe de nombreuses autres estimations plus rapides. Mais le sujet de ce TP n'est pas la performance mais d'améliorer le ressenti de l'utilisateur.

[Essayer le jeu sans les workers !](#)

Des problèmes

Comme vous pouvez le remarquer, il souffre de quelques défauts lorsque l'ordinateur prend du temps pour réfléchir (si sur votre ordinateur, cela ne prend pas beaucoup de temps, vous pouvez agrandir le plateau ou augmenter la profondeur de recherche). On ne sait pas combien de temps on doit encore attendre avant que l'ordinateur joue son coup.

Pire si le plateau est particulièrement grand ou si la profondeur de recherche est importante, un message apparaît demandant s'il faut interrompre le script.

De plus lorsqu'on clique sur une case pour voir si le jeu réagit, ce coup sera joué lorsque l'ordinateur rend la main...

Tout ceci n'est pas très joli, mais vous allez pouvoir l'améliorer en utilisant la technologie des workers 😊.

Correction

Avant de se lancer dans un code en intégrant les workers, il est toujours utile de commencer par réaliser un code qui fonctionne sans web-worker.

Tout d'abord cela nous permet de déboguer plus facilement le code.

Ensuite, puisque nous avons déjà écrit ce code, il est plus facile de proposer une solution alternative à ceux qui utilisent un navigateur ne supportant pas les web-workers.

Et enfin, une fois cette phase finie, on peut juger si les workers apporteront vraiment quelque chose au programme. On peut alors se rendre compte qu'il est inutile de rendre son code plus complexe (car cela le rendra aussi plus difficile à maintenir) et de s'arrêter là.

Dans le cas de ce TP, le code sans worker était déjà fourni (à moins que vous ne l'ayez écrit vous-même 😎) et la liste de ses défauts a déjà été exposée.

Nous allons donc nous atteler à la tâche pour ajouter un worker.

Explication de la mise en place d'un worker

Tout d'abord il est important de cerner les fonctions qui devront être externalisées dans le worker. Pour cela il faut avoir une idée précise de la manière dont il sera utilisé.

Ensuite on pourra effectuer les modifications pour rendre le code compatible avec un worker.

Et enfin créer toutes les communications utiles à chaque thread.

Analyse des besoins

Dans ce TP, le problème principal est le temps de réflexion de l'IA. Comme le but n'est pas d'améliorer la performance de l'IA (ceci serait un sujet d'algorithmie), nous allons donc placer toute la partie réflexion de l'IA dans un worker. Ainsi le thread principal sera libéré et l'utilisateur ne se retrouvera pas devant une page figée ou avec un message l'incitant à interrompre le script.

Pour faire fonctionner cette IA dans un thread séparé, il faudra lui envoyer la grille de jeu (à partir de laquelle elle doit trouver le meilleur coup), les règles et les options (profondeur de recherche, et le nombre de pions à aligner), et bien sûr la couleur avec laquelle elle doit maintenant jouer.

Ainsi la séparation est déjà faite : le fichier `grille.js` sera le thread principal, et le fichier `IA.js` correspondra à notre fichier de base pour le worker.

Si vous regardez bien la fonction `iaAlphaBeta`, vous pouvez remarquer qu'elle fait appel à la fonction `verifVainqueur` afin de voir si un coup donne un gagnant. Or cette fonction fait partie du fichier `grille.js` et est utilisée dans celui-ci (dans la fonction `joue`).



Mais alors que faire ?

La première idée qu'on a est de copier cette fonction dans le fichier `IA.js`. Certes cela fonctionnera.

Mais imaginons que plus tard, vous désireriez changer les règles. Cela peut arriver si vous avez remarqué une erreur, ou si vous voulez ajouter des options (par exemple, un joueur ne gagne que s'il a strictement 5 pions alignés).

Il ne faudra pas oublier d'effectuer vos modifications dans les deux fonctions !

Rappelez-vous ce que vous avez appris dans le chapitre précédent : `importScripts`.

Grâce à `importScripts` on pourra écrire la fonction dans un fichier et l'appeler dans chaque thread.

Ainsi si une modification doit être apportée, il ne faudra l'effectuer qu'une seule fois ! 🎉

On va donc déplacer la fonction `verifVainqueur` dans un fichier `verifFin.js`.

Il ne faut pas oublier d'ajouter l'appel à ce fichier dans le fichier `html` :

Code : HTML

```
<script src="./verifFin.js"></script>
```

Préparation du worker

Maintenant que la stratégie est prête, place à l'action !

Tout d'abord, il va falloir enlever l'appel au fichier IA.js.

Code : HTML

```
<script src="./IA.js"></script>
```

Dans le thread principal (donc le fichier grille.js), on ajoute une variable globale qui contiendra l'instance du worker :

Code : JavaScript - grille.js

```
var iaWorker; // worker gérant l'IA (si le navigateur supporte les  
workers)
```

Le but est aussi de proposer une solution aux utilisateurs ayant un navigateur ne supportant pas les workers. Il suffit de tester l'existence du constructeur Worker et de proposer un contournement sinon.

Le contournement consiste simplement à exécuter le script IA.js dynamiquement :

Code : JavaScript - grille.js

```
if(window.Worker){  
  iaWorker = new Worker("./IA.js");  
}else{  
  //solution du substitution au worker  
  iaWorker = document.createElement("script");  
  iaWorker.src = "./IA.js";  
  document.body.appendChild(iaWorker);  
  iaWorker = null; //afin que iaWorker ne soit définit QUE si le  
  worker existe  
}
```

Comme on souhaite ajouter une information concernant la progression de la recherche de l'IA, on ajoute également une variable globale qui contiendra une référence à l'élément HTML donnant cette information.

Code : JavaScript - grille.js

```
var progressIA; //élément permettant d'indiquer où en est  
l'ordinateur
```

Il reste maintenant à créer cet élément et le lier au document. Ainsi dans la fonction affichageDOM, on ajoute :

Code : JavaScript - grille.js

```
//barre de progression indiquant l'avancée de la recherche de  
l'ordinateur  
progressIA = document.createElement("progress");  
progressIA.max = 100;  
elementIA.appendChild(progressIA);
```

Pour l'instant aucune valeur n'est définie, afin que la barre utilise le comportement par défaut jusqu'à ce qu'on définisse explicitement une valeur. Le maximum a été défini à 100, il faudra donc définir l'état d'avancement en pourcentage.

Il ne faut donc pas oublier que maintenant le fichier IA.js peut être appelée de deux manières : soit à l'intérieur d'un worker, soit de manière traditionnelle.

Et il doit pouvoir fonctionner correctement dans les deux cas !

Ne vous en faites pas, ce ne sera pas si difficile.

Pour que le code puisse fonctionner dans un worker, il lui manque encore une chose : l'appel à la fonction `verifVainqueur` qui se trouve maintenant dans le fichier `verifFin.js`.

Dans le cas où `IA.js` est appelé hors worker, la fonction `verifVainqueur` est déjà accessible puisque le fichier a été chargé dans le HTML.

Il reste donc à charger le fichier dans le cas du worker. En sachant que `importScripts` est une fonction qui n'existe que dans un worker, on peut en profiter pour définir une variable nous indiquant cet état (ce qui facilitera cette détection ultérieurement) :

Code : JavaScript - IA.js

```
if(typeof importScripts === "function"){
  //dans le cas d'un worker, on importe le script permettant de
  vérifier la fin d'une partie
  importScripts("../verifFin.js");
  self.inWorker = true;
}else{
  window.inWorker = false;
}
```

Voilà maintenant tout le travail préparatoire est terminé. Dans le cas où le navigateur ne supporte pas `Worker`, le programme doit toujours fonctionner normalement. Pour tester ce bon fonctionnement, il suffit de mal orthographier `window.Worker`. Vous pouvez ainsi passer en mode sans worker et vérifier que cela fonctionne toujours correctement.

Gestion des communications

Actuellement si on teste le fonctionnement de la page avec le `Worker`, aucune erreur ne devrait apparaître, mais l'IA ne fonctionnera pas car il n'y a actuellement aucune communication.

Le thread principal n'enverra qu'un seul type de demande : quel coup l'IA désire jouer ?

Le worker va envoyer deux types de messages :

- Lorsque l'IA indique quel coup est le meilleur. Ce qui indique aussi que le worker a fini de travailler.
- Lors d'une information sur l'avancée de la recherche, qui consistera à mettre à jour la barre de progression

Pour distinguer ces messages, on peut les intégrer à un objet et définir un attribut "commande" avec un mot clef.

Préparons la réception des messages dans le thread principal.

Au niveau de la création du worker, on peut donc écrire le listener message du worker :

Code : JavaScript - grille.js

```
iaWorker.onmessage = function(e){
  //réception des messages du workers
  var data = e.data;
  switch(data.cmd){
    case "update":
      progressIA.value = data.value;
      break;
    case "coup":
      continueJeu = true;
      elementIA.style.visibility = "hidden";
      joue(data.x,data.y);
      break;
  }
};
```

Pour réaliser la mise à jour, il faut donc envoyer un objet ayant un attribut `cmd` ayant pour valeur `"update"` et un attribut `value` ayant pour valeur l'avancée de la recherche en pourcentage.

Ce message peut être envoyé depuis la fonction `iaAlphaBeta`.

Afin d'éviter d'envoyer ce message trop fréquemment, cette information ne sera envoyée que lorsque la profondeur de recherche est à 0, c'est à dire lorsque l'IA explore un nouveau coup qu'elle est susceptible de proposer.

Code : JavaScript - IA.js

```
//on va essayer toutes les combinaisons possible
for(var x=0;x<nx;x++){
  for(var y=0;y<ny;y++){
    if(grille[x][y]) continue; //case déjà occupée
    if(!profondeur && inWorker){
      postMessage({cmd:"update",value:(x*ny+y)*100/(nx*ny)});
    }
    ...
  }
}
```

Dans le worker, la réalisation du listener "message" permet d'initialiser toutes les variables utilisées pour la recherche. Une fois celle-ci finie, il ne reste plus qu'à envoyer le résultat selon le modèle défini précédemment.

Code : JavaScript - IA.js

```
//réception des messages
onmessage = function(e) {
  var data = e.data;
  self.iaProfondeurMax = data.profondeur;
  self.nx=data.grille.length;
  self.ny=data.grille[0].length;
  self.nbAligne=data.nbAligne;
  var coup = iaAlphaBeta(data.grille, data.tour, 0, -Infinity,
  Infinity);
  postMessage({cmd:"coup",x:coup[0],y:coup[1]});
};
```

L'écriture de ce listener permet de montrer exactement les besoins du worker pour fonctionner. Il faut lui envoyer un objet avec les attributs suivants : grille, profondeur, nbAligne, et tour.

Nous pouvons donc maintenant écrire l'envoi du message qui déclenchera la mise en route de l'IA. Dans la fonction iaToPlay, il est possible de modifier le déclenchement de l'IA de cette manière :

Code : JavaScript - grille.js

```
if(iaWorker){
  iaWorker.postMessage({grille:grille,tour:couleurTour,profondeur:iaProfondeurMax
};
}else{
  setTimeout(function(){
    var rslt = iaJoue(grille,couleurTour);
    continueJeu = true;
    elementIA.style.visibility = "hidden";
    joue(rslt[0],rslt[1]);
  },10); //au cas où deux ordi jouent ensemble et pour voir le coup pendant que
}
```

À ce stade, le programme répond à l'énoncé : l'utilisateur n'est plus figé sur la page pendant que l'IA réfléchit et il est informé de l'avancée de la recherche.

Toutefois il reste quelques détails qui posent problème, c'est ce que nous allons voir maintenant.

Analyse des problèmes d'actions simultanées

Imaginez que l'IA est en train de fonctionner, et qu'à ce moment l'utilisateur clique sur le bouton Commencer, que se passe-t-il ?

Le jeu est réinitialisé, et une nouvelle grille est affichée. Et quand le worker a fini son travail, un coup est ajouté ! Alors que ce n'était même pas son tour !

Il s'agit d'un problème qui arrive parce que plusieurs tâches peuvent être exécutées en même temps.

Il y a aussi un autre cas à étudier : lorsque l'utilisateur décide qu'une couleur n'est plus jouée par l'ordinateur.



Comment résoudre ces problèmes ?

Tout d'abord, il devient intéressant de savoir s'il y a une recherche en cours ou non. Pour cela on va attacher une variable playing à notre objet iaWorker.

Il faut donc bien mettre à jour cette variable quand on démarre l'IA (dans iaToPlay).

Code : JavaScript - grille.js

```
if(iaWorker){
  iaWorker.playing = true;
  iaWorker.postMessage({grille:grille,tour:couleurTour,profondeur:iaProfondeurMax,
};
}
```

Mais il faut aussi la mettre à jour quand celle-ci a fini son travail (dans le listener de "message").

Code : JavaScript - grille.js

```
case "coup":
  continueJeu = true;
  elementIA.style.visibility = "hidden";
  iaWorker.playing = false;
  joue(data.x, data.y);
  break;
```

Maintenant qu'on sait si le worker travaille ou non, nous allons pouvoir corriger les problèmes.

Puisqu'on demande de recommencer une nouvelle partie, cela signifie que le travail du worker est devenu inutile. Il suffit donc de tuer le worker et d'en créer un nouveau qui sera prêt à être lancé.

Comme on souhaite créer le worker à différents endroits, autant créer une fonction dédiée à cette tâche.

Code : JavaScript - grille.js

```
//Création d'un worker pour l'IA
function createIAWorker(){
  iaWorker = new Worker("./IA.js"); //création du worker
  iaWorker.playing = false; // initialement le worker ne travaille pas

  iaWorker.onmessage = function(e){
    //réception des messages du workers
    var data = e.data;
    switch(data.cmd){
      case "update":
        progressIA.value = data.value;
        break;
      case "coup":
        continueJeu = true;
        elementIA.style.visibility = "hidden";
        iaWorker.playing = false;
        joue(data.x, data.y);
        break;
    }
  };

  iaWorker.onerror = function(e){
    //Gestion des erreurs
    alert(e.message);
  };
}
```

Dans la fonction init, on peut donc maintenant écrire :

Code : JavaScript - grille.js

```
if(iaWorker && iaWorker.playing){ //si le worker ne travaille pas inutile de le tuer
  iaWorker.terminate();
  createIAWorker();
  elementIA.style.visibility = "hidden";
}
```

Pour ce qui est des changements de joueur humain/IA, il y a plusieurs choix possibles :

1. l'IA continue de travailler, le changement ne sera effectif qu'au prochain tour.
2. l'humain peut jouer, mais l'IA continue de réfléchir au cas où la case serait à nouveau cochée.
3. l'humain peut jouer et le travail de l'IA est interrompu. Mais si la case est à nouveau cochée, l'IA doit reprendre de zéro.

Par simplicité, je vais choisir le premier cas. Pourquoi par simplicité ? Simplement par ce qu'il n'y a rien à faire !

Pour finir une petite amélioration, dans le worker, comme la grille a déjà été copiée lors de l'envoi du message (objet cloné), on peut optimiser légèrement l'IA en évitant la copie de la grille :

Code : JavaScript - IA.js

```
function iaJoue(grilleOrig, couleur){
```

```
var grille = inWorker?grilleOrig:copieGrille(grilleOrig);
return iaAlphaBeta(grille, couleur, 0, -Infinity, Infinity);
}
```

Solution avec le worker

En effectuant toutes ces modifications, vous devriez maintenant obtenir une page qui répond aux critères de l'énoncé. Voici les codes qui contiennent ces modifications :

Secret ([cliquez pour afficher](#))

Code : HTML

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8"/>
<title>TP : jeu du Morpion</title>
<style>
table{
background-color:#FFCC66;
}
td{
border: 1px solid black;
border-radius: 25px;
width: 25px;
height:25px;
cursor:pointer;
}
td.empty{
background-color:inherit;
}
td.noir{
background-color:#000000;
box-shadow: inset 4px 3px 20px -6px #146717;
}
td.blanc{
background-color:#EEEEFF;
box-shadow: inset 4px 3px 20px -6px #7f8f8f;
}
fieldset{
display: inline-block;
}
fieldset label{
display: block;
}
</style>
</head>
<body>
<script src="./grille.js"></script>
<script src="./verifFin.js"></script>
</body>
</html>
```

Code : JavaScript - grille.js

```
var nx = 7; //nombre de cellules en largeur
var ny = 7; //nombre de cellules en hauteur
var nbAligne = 5; //nombre de jetons à aligner pour gagner
var couleurTour = 1; //couleur dont c'est le tour
var continueJeu = false; //permet d'indiquer si le jeu est arrêté ou non

var iaProfondeurMax = 4; //indique la profondeur de recherche de l'IA
var iaNoir = false; //indique si le joueur noir est une IA
var iaBlanc = true; //indique si le joueur blanc est une IA

var grille = []; //grille du jeu
var iaWorker; // worker gérant l'IA (si le navigateur supporte les workers)

var elemTable; //élément contenant les éléments d'affichage du jeu
var elemIA; //élément indiquant que l'ordinateur réfléchit
var progressIA; //élément permettant d'indiquer où en est l'ordinateur

//Affichage des éléments pour le paramétrage
function affichageDOM(){
//Règles
var fieldset = document.createElement("fieldset");
var legend = document.createElement("legend");
legend.textContent = "Règles";
```

```

fieldset.appendChild(legend);

//NX
var label = document.createElement("label");
label.textContent = "Largeur :";
var inputNX = document.createElement("input");
inputNX.type="number";
inputNX.min=1;
inputNX.value=nx;
label.appendChild(inputNX);
fieldset.appendChild(label);

//NY
label = document.createElement("label");
label.textContent = "Hauteur :";
var inputNY = document.createElement("input");
inputNY.type="number";
inputNY.min=1;
inputNY.value=ny;
label.appendChild(inputNY);
fieldset.appendChild(label);

//aligne
label = document.createElement("label");
label.textContent = "Nombre de jetons à aligner pour gagner :";
var inputAlign = document.createElement("input");
inputAlign.type="number";
inputAlign.min=1;
inputAlign.value=nbAligne;
label.appendChild(inputAlign);
fieldset.appendChild(label);

document.body.appendChild(fieldset);

//Pour l'IA
fieldset = document.createElement("fieldset");
legend = document.createElement("legend");
legend.textContent = "configuration de l'IA";
fieldset.appendChild(legend);

//IA noir?
label = document.createElement("label");
label.textContent = "Le joueur noir est un ordinateur :";
input = document.createElement("input");
input.type="checkbox";
input.checked=false;
input.onchange=function(){
  iaNoir=this.checked;
  iaToPlay(); //on vérifie si c'est au tour de l'IA de jouer
};
label.appendChild(input);
fieldset.appendChild(label);

//IA blanc?
label = document.createElement("label");
label.textContent = "Le joueur blanc est un ordinateur :";
input = document.createElement("input");
input.type="checkbox";
input.checked=true;
input.onchange=function(){
  iaBlanc=this.checked;
  iaToPlay(); //on vérifie si c'est au tour de l'IA de jouer
};
label.appendChild(input);
fieldset.appendChild(label);

//Profondeur
label = document.createElement("label");
label.textContent = "Profondeur de recherche :";
input = document.createElement("input");
input.type="number";
input.min=1;
input.value=iaProfondeurMax;
input.onchange=function(){iaProfondeurMax=parseInt(this.value,10);};
label.appendChild(input);
fieldset.appendChild(label);

document.body.appendChild(fieldset);

//bouton permettant de lancer la partie
var btnStart = document.createElement("button");
btnStart.textContent = "Commencer";
btnStart.onclick=function(){
  nx = parseInt(inputNX.value,10);
  ny = parseInt(inputNY.value,10);
  nbAligne = parseInt(inputAlign.value,10);
  init();
}

```



```

document.body.appendChild(btnStart);

//Indicateur que l'ordinateur réfléchit
elementIA = document.createElement("div");
elementIA.textContent = "L'ordinateur est en train de réfléchir...";
elementIA.style.visibility = "hidden";

//barre de progression indiquant l'avancée de la recherche de l'ordinateur
progressIA = document.createElement("progress");
progressIA.max = 100;
elementIA.appendChild(progressIA);
document.body.appendChild(elementIA);

document.body.appendChild(document.createElement("hr"));
}

window.addEventListener("load",affichageDOM,false);

//Initialisation d'une partie
function init(){
  if(iaWorker && iaWorker.playing){
    //l'IA est encore en train de chercher une solution. On va donc l'arrêter
    iaWorker.terminate();
    createIAWorker();
    elementIA.style.visibility = "hidden";
  }

  //initialisation de la grille
  for(var x=0;x<nx;x++){
    grille[x] = [];
    for(var y=0;y<ny;y++){
      grille[x][y] = 0;
    }
  }

  //suppression de l'élément HTML de la grille précédente
  if(elemTable){
    document.body.removeChild(elemTable);
  }

  //affichage de la grille de jeu
  elemTable = document.createElement("table");
  var row,cel;
  for(y=0;y<ny;y++){
    row = elemTable.insertRow(-1);
    for(x=0;x<nx;x++){
      cel = row.insertCell(-1);
      cel.id = "grille"+x+"_"+y;
      cel.onclick=setClick(x,y);
      switch(grille[x][y]){
        case 1:
          cel.className = "noir";
          break;
        case 2:
          cel.className = "blanc";
          break;
        case 0:
        default:
          cel.className = "empty";
      }
    }
  }
  document.body.appendChild(elemTable);
  couleurTour = 1;
  continueJeu = true;
  iaToPlay(); //on vérifie si c'est au tour de l'IA de jouer
};

//permet de changer l'affichage de la couleur d'un jeton
function changeCouleur(x,y){
  grille[x][y]=couleurTour;
  var elem = document.getElementById("grille"+x+"_"+y);
  if(elem){
    elem.className=couleurTour===1?"noir":"blanc";
  }
}

//permet de jouer un coup en x,y
function joue(x,y){
  if(!continueJeu) return false;
  if(grille[x][y]) return false;
  var rslt;
  changeCouleur(x,y);
  couleurTour = couleurTour%2+1;
  if(rslt=verifVainqueur(x,y)){ //y a-t-il un gagnant ?
    continueJeu = false;
    alert((rslt===1?"Noirs":"Blancs")+" vainqueurs");
  }
}

```

```

    }

    if(!verifNbLibre()){
        continueJeu = false;
        alert("Parie nulle : égalité");
    }

    //est-ce que le prochain coup doit être joué par l'IA ?
    iaToPlay();
}

//Permet de vérifier si le prochain coup doit être joué par l'IA
function iaToPlay(){
    if(!continueJeu) return false;
    if((couleurTour === 1 && iaNoir) || (couleurTour === 2 && iaBlanc)){
        continueJeu = false; //pour empêcher un humain de jouer
        elementIA.style.visibility = "visible";
        if(iaWorker){
            iaWorker.playing = true;
            iaWorker.postMessage({grille:grille,tour:couleurTour,profondeur:iaProfondeur});
        }else{
            setTimeout(function(){
                var rslt = iaJoue(grille,couleurTour);
                continueJeu = true;
                elementIA.style.visibility = "hidden";
                joue(rslt[0],rslt[1]);
            },10); //au cas où deux ordi jouent ensemble et pour voir le coup pendant
        }
    }
}

//permet de créer une fonction listener sur un élément x,y
function setClick(x,y){
    return function(){
        joue(x,y);
    };
}

//permet de vérifier s'il reste des coups jouable
function verifNbLibre(){
    var nbLibre=0;
    for(var x=0;x<nx;x++){
        for(var y=0;y<ny;y++){
            if(grille[x][y]===0){
                nbLibre++;
            }
        }
    }
    return nbLibre;
}

//Création d'un worker pour l'IA
function createIAWorker(){
    iaWorker = new Worker("./IA.js"); //création du worker
    iaWorker.playing = false;

    iaWorker.onmessage = function(e){
        //réception des messages du workers
        var data = e.data;
        switch(data.cmd){
            case "update":
                progressIA.value = data.value;
                break;
            case "coup":
                continueJeu = true;
                elementIA.style.visibility = "hidden";
                iaWorker.playing = false;
                joue(data.x,data.y);
                break;
        }
    };

    iaWorker.onerror = function(e){
        //Gestion des erreurs
        alert(e.message);
    };
}

if(window.Worker){
    createIAWorker();
}else{
    //solution du substitution au worker
    iaWorker = document.createElement("script");
    iaWorker.src = "./IA.js";
    document.body.appendChild(iaWorker);
    iaWorker = null; //afin que iaWorker ne soit défini QUE si le worker exist
}

```

Code : JavaScript - verifFin.js

```
//permet de vérifier s'il y a un vainqueur (en ne regardant que le dernier coup joué)
function verifVainqueur(x,y,vGrille){
  vGrille = vGrille || grille;
  var col = vGrille[x][y]; //couleur du jeton qui vient d'être joué
  var alignH = 1; //nombre de jetons alignés horizontalement
  var alignV = 1; //nombre de jetons alignés verticalement
  var alignD1 = 1; //nombre de jetons alignés diagonalement NO-SE
  var alignD2 = 1; //nombre de jetons alignés diagonalement SO-NE
  var xt,yt;

  //vérification horizontale
  xt=x-1;
  yt=y;
  while(xt>=0 && vGrille[xt][yt]==col){
    xt--;
    alignH++;
  }
  xt=x+1;
  yt=y;
  while(xt<nx && vGrille[xt][yt]==col){
    xt++;
    alignH++;
  }

  //vérification verticale
  xt=x;
  yt=y-1;
  while(yt>=0 && vGrille[xt][yt]==col){
    yt--;
    alignV++;
  }
  xt=x;
  yt=y+1;
  while(yt<ny && vGrille[xt][yt]==col){
    yt++;
    alignV++;
  }

  //vérification diagonale NO-SE
  xt=x-1;
  yt=y-1;
  while(xt>=0 && yt>=0 && vGrille[xt][yt]==col){
    xt--;
    yt--;
    alignD1++;
  }
  xt=x+1;
  yt=y+1;
  while(xt<nx && yt<ny && vGrille[xt][yt]==col){
    xt++;
    yt++;
    alignD1++;
  }

  //vérification diagonale SO-NE
  xt=x-1;
  yt=y+1;
  while(xt>=0 && yt<ny && vGrille[xt][yt]==col){
    xt--;
    yt++;
    alignD2++;
  }
  xt=x+1;
  yt=y-1;
  while(xt<nx && yt>=0 && vGrille[xt][yt]==col){
    xt++;
    yt--;
    alignD2++;
  }

  //parmi tous ces résultats on regarde s'il y en a un qui dépasse le nombre nécessaire pour gagner
  if(Math.max(alignH,alignV,alignD1,alignD2)>=nbAligne){
    return col;
  }else{
    return 0;
  }
}
```

Code : JavaScript - IA.js

```

if(typeof importScripts === "function"){
  //dans le cas d'un worker, on importe le script permettant de
  vérifier la fin d'une partie
  importScripts("./verifFin.js");
  //si importScripts existe c'est qu'on est dans un worker. On va
  en profiter pour définir une variable afin de faciliter la
  détection ultérieure
  self.inWorker = true;
}else{
  window.inWorker = false;
}

//réception des messages
onmessage = function(e){
  var data = e.data;
  self.iaProfondeurMax = data.profondueur;
  self.nx=data.grille.length;
  self.ny=data.grille[0].length;
  self.nbAligne=data.nbAligne;
  var coup = iaJoue(data.grille,data.tour);
  postMessage({cmd:"coup",x:coup[0],y:coup[1]});
};

//demande à l'IA de jouer
function iaJoue(grilleOrig,couleur){
  var grille = inWorker?grilleOrig:copieGrille(grilleOrig);
  return iaAlphaBeta(grille, couleur, 0, -Infinity, Infinity);
}

//fonction gérant l'algorithme minimax et l'élégage alpha-beta
function iaAlphaBeta(grille, couleur, profondeur, alpha, beta){
  if(profondueur === iaProfondeurMax){
    //on a atteint la limite de profondeur de calcul on retourne
    donc une estimation de la position actuelle
    if(couleur === 1){
      return iaEstimation(grille);
    }else{
      return -iaEstimation(grille);
    }
  }else{
    var meilleur = -Infinity; //estimation du meilleur coup actuel
    var estim; //estimation de la valeur d'un coup
    var coup=null; //meilleur coup actuel
    var couleurOpp = couleur%2+1; //optimisation pour calculer la
    couleur adverse

    //on va essayer toutes les combinaisons possible
    for(var x=0;x<nx;x++){
      for(var y=0;y<ny;y++){
        if(grille[x][y]) continue; //case déjà occupée
        if(!profondeur && inWorker){
          postMessage({cmd:"update",value: (x*ny+y)*100/(nx*ny)});
        }

        if(!coup){coup=[x,y];} //pour proposer au moins un coup

        grille[x][y]=couleur; //on va essayer ce coup
        //vérifie si le coup est gagnant
        if(estim=verifVainqueur(x,y,grille)){
          grille[x][y]=0; //restauration de la grille
          if(!profondeur){
            return [x,y];
          }else{
            return Infinity;
          }
        }
        estim = -iaAlphaBeta(grille, couleurOpp, profondeur+1, -beta,
        -alpha); //on calcule la valeur de ce coup

        if(estim > meilleur){
          //on vient de trouver un meilleur coup
          meilleur = estim;
          if(meilleur > alpha){
            alpha = meilleur;
            coup = [x,y];
            if(alpha >= beta){
              //ce coup est mieux que le meilleur des coups qui auraient
              put être joués si on avait joué un autre
              coup. Cela signifie que jouer le coup qui a amené à cette
              position n'est pas bon. Il est inutile
              de continuer à estimer les autres possibilités de cette position
              (principe de l'élégage alpha-beta). */
              grille[x][y]=0; //restauration de la grille
              if(!profondeur){

```

```

        if(!profondeur){
            return coup;
        }else{
            return meilleur;
        }
    }
}
grille[x][y]=0; //restauration de la grille
}
if(!profondeur){
    return coup;
}else{
    if(coup) return meilleur;
    else return 0; //si coup n'a jamais été défini c'est qu'il n'y
    plus de possibilité de jeu. C'est partie nulle.
}
}
}

//permet d'estimer la position
function iaEstimation(grille){
    var estimation = 0; //estimation global de la position

    for(var x=0;x<nx;x++){
        for(var y=0;y<ny;y++){
            if(!grille[x][y]) continue;
            //estimation de la valeur de ce jeton et ajout au calcul
            d'estimation global
            switch(grille[x][y]){
                case 1:
                    estimation += iaAnalyse(grille,x,y);
                    break;
                case 2:
                    estimation -= iaAnalyse(grille,x,y);
                    break;
            }
        }
    }
    return estimation;
}

//permet de calculer le nombre de "liberté" pour la case donnée
function iaAnalyse(grille,x,y){
    var couleur = grille[x][y];
    var estimation = 0; //estimation pour toutes les directions
    var compteur = 0; //compte le nombre de possibilité pour une
    direction
    var centre = 0; //regarde si le jeton a de l'espace de chaque
    côté
    var bonus = 0; //point bonus lié aux jetons alliés dans cette
    même direction
    var i,j; //pour les coordonnées temporaires
    var pass=false; //permet de voir si on a dépassé la case étudiée
    var pliberte = 1; //pondération sur le nombre de liberté
    var pBonus = 1; //pondération Bonus
    var pCentre = 2; //pondération pour l'espace situé de chaque côté
    é

    //recherche horizontale
    for(i=0;i<nx;i++){
        if(i==x){
            centre = compteur++;
            pass=true;
            continue;
        }
        switch(grille[i][y]){
            case 0: //case vide
                compteur++;
                break;
            case couleur: //jeton allié
                compteur++;
                bonus++;
                break;
            default: //jeton adverse
                if(pass){
                    i=nx; //il n'y aura plus de liberté supplémentaire, on
                    arrête la recherche ici
                }else{
                    //on réinitialise la recherche
                    compteur = 0;
                    bonus = 0;
                }
            }
        }
    }
    if(compteur>=nbAligne){
        //il est possible de gagner dans cette direction
    }
}

```

```

    estimation += compteur*pLiberte + bonus*pBonus +
    (1-Math.abs(centre/(compteur-1)-0.5))*compteur*pCentre;
}

//recherche verticale
compteur=0;
bonus=0;
pass=false;
for(j=0;j<ny;j++){
    if(j==y){
        centre=compteur++;
        pass=true;
        continue;
    }
    switch(grille[x][j]){
        case 0: //case vide
            compteur++;
            break;
        case couleur: //jeton allié
            compteur++;
            bonus++;
            break;
        default: //jeton adverse
            if(pass){
                j=ny; //il n'y aura plus de liberté supplémentaire, on
                arrête la recherche ici
            }else{
                //on réinitialise la recherche
                compteur = 0;
                bonus = 0;
            }
    }
}
if(compteur>=nbAligne){
    //il est possible de gagner dans cette direction
    estimation += compteur*pLiberte + bonus*pBonus +
    (1-Math.abs(centre/(compteur-1)-0.5))*compteur*pCentre;
}

//recherche diagonale (NO-SE)
compteur=0;
bonus=0;
i=x;
j=y;
while(i-->0 && j-->0){
    switch(grille[i][j]){
        case 0: //case vide
            compteur++;
            break;
        case couleur: //jeton allié
            compteur++;
            bonus++;
            break;
        default: //jeton adverse, on arrete de rechercher
            i=0;
    }
}
centre=compteur++;
i=x;
j=y;
while(++i<nx && ++j<ny){
    switch(grille[i][j]){
        case 0: //case vide
            compteur++;
            break;
        case couleur: //jeton allié
            compteur++;
            bonus++;
            break;
        default: //jeton adverse, on arrete de rechercher
            i=nx;
    }
}
if(compteur>=nbAligne){
    //il est possible de gagner dans cette direction
    estimation += compteur*pLiberte + bonus*pBonus +
    (1-Math.abs(centre/(compteur-1)-0.5))*compteur*pCentre;
}

//recherche diagonale (NE-SO)
compteur=0;
bonus=0;
i=x;
j=y;
while(i-->0 && ++j<ny){
    switch(grille[i][j]){
        case 0: //case vide
            compteur++;

```

```

        break;
    case couleur: //jeton allié
        compteur++;
        bonus++;
        break;
    default: //jeton adverse, on arrete de rechercher
        i=0;
    }
}
centre=compteur++;
i=x;
j=y;
while(++i<nx && j-->0){
    switch(grille[i][j]){
        case 0: //case vide
            compteur++;
            break;
        case couleur: //jeton allié
            compteur++;
            bonus++;
            break;
        default: //jeton adverse, on arrete de rechercher
            i=nx;
    }
}
if(compteur>=nbAligne){
    //il est possible de gagner dans cette direction
    estimation += compteur*pLiberte + bonus*pBonus +
    (1-Math.abs(centre/(compteur-1)-0.5))*compteur*pCentre;
}

return estimation;
}

//permet de copier une grille, cela permet d'éviter de modifier
//par inadvertance la grille de jeu originale
function copieGrille(grille){
    var nvGrille=[];
    for(var x=0;x<nx;x++){
        nvGrille[x]=grille[x].concat([]); //effectue une copie de la
        liste
    }
    return nvGrille;
}

```

[Essayer la solution !](#)

Pour aller plus loin

Vous voulez encore améliorer ce programme ? Alors voici quelques idées de défi :

- Implémenter un autre comportement, lorsque l'utilisateur décide que ce n'est plus à l'IA de jouer. Vous pouvez implémenter le 2e (le plus complet) ou le 3e cas.
- Améliorer la performance de l'IA.

Il faudra sans doute reprendre l'algorithme et améliorer la performance de l'estimation. Ensuite il est possible de construire une autre architecture où l'IA commence à chercher pendant que ce n'est pas son tour. La difficulté consistera à réaffiner au bon moment la recherche lorsque l'adversaire joue son coup. Sinon il peut être possible de construire une architecture où les coups calculés lors d'une précédente recherche ne sont pas oubliés.
- Ajouter des options aux règles. Il existe plusieurs variantes pour jouer au gomoku. Voici quelques variantes :
 - Il faut aligner strictement cinq pions pour gagner. Si un joueur aligne 6 pions, il n'a pas gagné.
 - Le joueur noir n'a pas le droit d'avoir deux séries de 3 pions alignés (où les deux extrémités sont libres).
 - Le joueur noir n'a pas le droit d'avoir deux séries de 4 pions alignés (où l'une des extrémités est libre).
 - Permettre des captures et déclarer vainqueur celui qui réussit à réaliser 5 captures.

Une capture peut être considérée lorsqu'en jouant un coup à la tête de deux pions adverse, si à l'autre bout de ces deux pions se trouvent aussi un de nos pions. Alors on peut capturer (retirer) les pions adverse. On peut aussi considérer une capture à la manière du jeu de Go : un pion (ou un groupe de pions) est(sont) retiré(s) s'il(s) est(sont) complètement entouré par des pions adverses.
- Dans une autre page, permettre d'avoir des informations détaillées de l'état de la recherche de coups. Comme avoir une carte d'estimation pour une position. Avoir l'estimation de la partie pour une profondeur de recherche donnée (surtout utile si la recherche est effectuée en "largeur d'abord").

Certaines de ces idées sont plutôt faciles à réaliser et d'autres beaucoup moins. Certaines demandent de réécrire complètement les workers, et d'autres juste de réfléchir à l'impact sur le code dans le worker.

Exercice supplémentaire : traitement d'image



Ce sujet est plutôt difficile si vous n'avez pas déjà un bon bagage technique, à la fois en javascript et en mathématiques. Toutefois vous pouvez vous contenter de la version sans worker et la modifier pour ajouter un worker.

Le sujet de cet exercice consiste à appliquer un filtre convolutif sur une image et afficher son résultat.
Le but sera d'informer l'utilisateur de la progression du traitement et aussi de permettre de démarrer plusieurs traitements en même temps.

La marche à suivre

Le but sera d'appliquer un traitement sur une image. Il nous faut donc d'abord une image et récupérer les valeurs de ses pixels. Pour cela je vous propose d'utiliser `canvas`. `Canvas` nous permet à la fois de réaliser un dessin (que ce soit l'utilisateur qui le crée ou une image qui est chargée) et de récupérer les pixels du `canvas`.



Si vous chargez une image dans Canvas, vous ne pourrez récupérer ses pixels que si cette image appartient au même domaine (principe du same origin policy)

Une fois que l'utilisateur demandera l'application d'un filtre, le traitement devra avoir lieu. Pendant ce temps l'utilisateur peut très bien demander un nouveau traitement ou modifier le dessin.

Une fois le traitement terminé, l'utilisateur doit pouvoir consulter le résultat.

Résumé

Voici les points à réaliser :

- Créer un dessin dans un canvas.
- Permettre à l'utilisateur de choisir son filtre.
- Réaliser le traitement de l'image :
 - Récupérer les pixels du Canvas.
 - Récupérer le filtre.
 - Appliquer le filtre sur l'image pour créer une nouvelle image.

L'utilisateur doit être informé de l'avancement du traitement.

- Afficher le résultat.

Quelques explications

En traitement d'image, les filtres permettent de mettre en valeur certaines propriétés intéressantes de l'image ou de donner un style à l'image.

Les filtres sont définis par des tableaux en deux dimensions représentant les valeurs à appliquer aux pixels voisins.

Par exemple, pour appliquer un flou à l'image, il faut effectuer une moyenne des pixels environnants. On appliquera alors un filtre moyenneur :

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

Une convolution est une opération consistant à appliquer un motif sur tous les points d'un espace.

Dans notre cas, l'espace correspond aux pixels de l'image (un tableau en deux dimensions). Et le motif est le filtre (qui est aussi un tableau en deux dimensions).

Plus simplement, cela consiste à effectuer un calcul pour chaque pixel de l'image. Ce calcul correspond à une somme des produits des valeurs du filtre avec celles des pixels avoisinant notre pixel cible.

Solution sans worker

Secret (cliquez pour afficher)

Code : HTML-index.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8"/>
<title>Traitement d'image</title>
<style>
  body{
    height:100%;
  }
  canvas{
    border: 1px solid #000000;
    z-index:1;
  }
</style>
</head>
<body>
```



```

    margin : 0px;
  }
  menu{
    height: 70px;
    margin : 0px;
    padding : 10px;
    margin-right : 300px;
  }
  .filteredCanvas{
    width : 50px;
    height : 25px;
  }
  .zoom{
    position : fixed;
    top : 98px;
    bottom : 10px;
    left : 8px;
    right : 300px;
    display : none;
  }
  .traitement{
    width:300px;
    position:absolute;
    right:0px;
    top:10px;
  }
</style>
</head>
<body>
<script src="./canvas.js"></script>
<script src="./gestionFiltre.js"></script>
</body>
</html>

```

Code : JavaScript - canvas.js

```

/*
 * création de la zone d'interaction
 */

//création d'une zone d'affichage pour les options liées au Canvas
var outils = document.createElement("menu");
document.body.appendChild(outils);

//création du canvas
var canvas = document.createElement("canvas");
canvas.addEventListener("mousedown",function(event){
  canvas.addEventListener("mousemove",draw,false);
  draw(event);
},false);
canvas.addEventListener("mouseup",function(event){
  canvas.removeEventListener("mousemove",draw,false);
},false);
canvas.addEventListener("mouseout",function(event){
  canvas.removeEventListener("mousemove",draw,false);
},false);
canvas.width = window.innerWidth - 320; //largeur du canvas
canvas.height = window.innerHeight - 120; //hauteur du canvas
document.body.appendChild(canvas);

//création des options de modification du pinceau
var elem_couleur = document.createElement("input");
elem_couleur.type = "color";
elem_couleur.value = "#000000";
elem_couleur.addEventListener("change",function(){ctx.fillStyle=this.value;},
outils.appendChild(elem_couleur);

var elem_taille = document.createElement("input");
elem_taille.type = "number";
elem_taille.min = 1;
elem_taille.value = 20;
elem_taille.title = "Taille du pinceau";
outils.appendChild(elem_taille);

//création d'un bouton de reset
var elem_reset = document.createElement("button");
elem_reset.textContent = "Reset";
elem_reset.addEventListener("click",erase,false);
outils.appendChild(elem_reset);

/**
 * Fonctions de gestion du Canvas
 */
function erase(){

```

```

ctx.save();
ctx.fillStyle="#FFFFFF";
ctx.fillRect(0, 0, canvas.width, canvas.height);
ctx.restore();
}

//permet de dessiner dans le canvas au niveau du curseur
function draw(event){
    var x = event.offsetX, //on récupère la position de la souris sur le canvas
        y = event.offsetY;
    ctx.beginPath();
    ctx.arc(x,y,elem_taille.value,0,Math.PI*2,true); //on trace un cercle
    ctx.fill();
}

/*
 * Initialisation
 */
var ctx = canvas.getContext("2d"); //référence au contexte de dessin
ctx.fillStyle = elem_couleur.value;

erase();

```

Code : JavaScript - gestionFiltre.js

```

/*
 * création de la zone d'interaction
 */

//Création d'une zone pour réaliser des traitements sur l'image
var elem_zoneTraitement = document.createElement("aside");
elem_zoneTraitement.className = "traitement";
document.body.appendChild(elem_zoneTraitement);

//ajout d'une liste de filtre
elem_zoneTraitement.appendChild( document.createTextNode("Filtre :
") );
var elem_listeFiltre = document.createElement("select");
elem_zoneTraitement.appendChild(elem_listeFiltre);

//ajout du bouton de démarrage
var btn_run = document.createElement("button");
btn_run.textContent = "Appliquer ce filtre";
btn_run.addEventListener("click",prepareFiltre,false);
elem_zoneTraitement.appendChild(btn_run);

//création de la zone de résultat
var elem_result = document.createElement("table");
elem_zoneTraitement.appendChild(elem_result);

//création d'une zone d'affichage pour le zoom du résultat
var elem_zoom = document.createElement("div");
elem_zoom.className = "zoom";
elem_zoneTraitement.appendChild(elem_zoom);

//création d'un canvas pour voir le résultat des filtres
var elem_canvasZoom = document.createElement("canvas");
elem_canvasZoom.width = canvas.width;
elem_canvasZoom.height = canvas.height;
elem_zoom.appendChild(elem_canvasZoom);

/**
 * Fonctions de gestion des filtres
 */
var listeFiltre = [
{
    nom:"Flou (petit)",
    filtre:[
        [1/10,1/10,1/10],
        [1/10,2/10,1/10],
        [1/10,1/10,1/10]
    ]
},
{
    nom:"Flou (moyen)",
    filtre:[
        [1/26,1/26,1/26,1/26,1/26],
        [1/26,1/26,1/26,1/26,1/26],
        [1/26,1/26,2/26,1/26,1/26],
        [1/26,1/26,1/26,1/26,1/26],
        [1/26,1/26,1/26,1/26,1/26]
    ]
}

```

```

    }
  },
  {
    nom: "Flou (grand)",
    filtre: [
      [1/50, 1/50, 1/50, 1/50, 1/50, 1/50, 1/50],
      [1/50, 1/50, 1/50, 1/50, 1/50, 1/50, 1/50],
      [1/50, 1/50, 1/50, 1/50, 1/50, 1/50, 1/50],
      [1/50, 1/50, 1/50, 2/50, 1/50, 1/50, 1/50],
      [1/50, 1/50, 1/50, 1/50, 1/50, 1/50, 1/50],
      [1/50, 1/50, 1/50, 1/50, 1/50, 1/50, 1/50],
      [1/50, 1/50, 1/50, 1/50, 1/50, 1/50, 1/50]
    ]
  },
  {
    nom: "Flou Gaussien (moyen,  $\sigma=0.7$ )",
    filtre: [
      [0.0001, 0.002, 0.0055, 0.002, 0.0001],
      [0.002, 0.0422, 0.1171, 0.0422, 0.002],
      [0.0055, 0.1171, 0.3248, 0.1171, 0.0055],
      [0.002, 0.0422, 0.1171, 0.0422, 0.002],
      [0.0001, 0.002, 0.0055, 0.002, 0.0001]
    ]
  },
  {
    nom: "Filtre de Laplace (petit)",
    filtre: [
      [-1, -1, -1],
      [-1, 8, -1],
      [-1, -1, -1]
    ]
  },
  {
    nom: "Sobel (vertical)",
    filtre: [
      [-1, 0, 1],
      [-2, 0, 2],
      [-1, 0, 1]
    ]
  },
  {
    nom: "Sobel (horizontal)",
    filtre: [
      [-1, -2, -1],
      [0, 0, 0],
      [1, 2, 1]
    ]
  }
];

//permet de préparer la liste des filtres disponibles
function generateFilterList(){
  var i = 0,
  li = listeFiltre.length,
  option;
  do{
    option = document.createElement("option");
    option.textContent = listeFiltre[i].nom;
    option.value = i;
    elem_listeFiltre.add(option, null);
  }while(++i<li);
}

//permet de préparer tout ce qui est nécessaire pour appliquer le
//filtre sur le canvas
function prepareFiltre(){
  var idFiltre = elem_listeFiltre.value;

  var imageID = []; //contiendra la liste des pixels correspondant
  //à l'image du Canvas
  var imageData = ctx.getImageData(0, 0, canvas.width,
  canvas.height); //récupération des données binaires du Canvas
  if(imageData){
    // on a réussi à extraire les données du Canvas, dans ce cas on
    //remplit imagePixels
    imageID = imageData.data;
  }//si on n'a pas réussi à extraire les données, alors on laisse
  //le tableau vide.

  //génération d'une nouvelle ligne de résultat
  var ligne = elem_result.insertRow(-1);
  var cellule = ligne.insertCell(0);
  cellule.textContent = listeFiltre[idFiltre].nom;

  cellule = ligne.insertCell(1);
  cellule.id = "filtre_"+uid;
  cellule.textContent = "en cours de génération";
}

```

```

    var image2D = conversionImage(image1D,canvas.width); //prepare
    l'image en 2D (+RVB)
    image2D = appliquerFiltre(image2D, idFiltre, uid); //on applique
    le filtre à l'image
    finalisationFiltre(image2D,uid++); //on affiche le résultat
}

//permet de convertir un tableau de pixel 1D en 2D
function conversionImage(image1D,w,uid){
    //prepare l'image en 2D (+RVB)
    var image2D = [],
    i,
    x=0,
    y=0,
    li = image1D.length;

    for(i=0 ; i<li; i++){
        if(y===0){
            image2D[x]=[];
        }
        image2D[x][y]=[];
        image2D[x][y][0]=image1D[i++];
        image2D[x][y][1]=image1D[i++];
        image2D[x][y][2]=image1D[i++];
        if(++x>=w){
            x=0;
            y++;
        }
    }

    return image2D;
}

//permet d'appliquer le filtre sur l'image
function appliquerFiltre(image, idFiltre){
    var filtre = (listeFiltre[idFiltre] &&
    listeFiltre[idFiltre].filtre) || [[]], //récupère le filtre s'il
    existe ou alors génère un filtre vide
    imgX, //position X sur l'image
    imgY, //position Y sur l'image
    imgMaxX = image.length, // largeur de l'image
    imgMaxY = image[0].length, //hauteur de l'image
    fltX, //position X sur le filtre
    fltY, //position Y sur le filtre
    fltMaxX = filtre.length, //largeur du filtre
    fltMaxY = filtre[0].length, //hauteur du filtre
    fltOffsetX = (fltMaxX - 1)/2, //offset X à appliquer sur le
    filtre pour trouver le bon pixel sur l'image
    fltOffsetY = (fltMaxY - 1)/2, //offset Y à appliquer sur le
    filtre pour trouver le bon pixel sur l'image
    index, //sert à identifier la position par rapport à la liste
    des pixels
    sommeRouge, //valeur temporaire pour l'application du filtre sur
    un pixel
    sommeVert, //valeur temporaire pour l'application du filtre sur
    un pixel
    sommeBleu, //valeur temporaire pour l'application du filtre sur
    un pixel
    x, //index X temporaire pour chercher le bon pixel dans l'image
    y, //index Y temporaire pour chercher le bon pixel dans l'image
    imageFinale=[]; // liste des pixels finales

    //on parcourt tous les pixels de l'image
    imageX:for(imgX = 0; imgX<imgMaxX; imgX++){
        imageFinale[imgX] = [];
        imageY:for(imgY = 0; imgY<imgMaxY; imgY++){
            sommeRouge = 0;
            sommeVert = 0;
            sommeBleu = 0;

            //on parcourt toutes les valeurs du filtre
            filtreX:for(fltX = 0; fltX<fltMaxX; fltX++){
                x = imgX + fltX - fltOffsetX; //on calcule la valeur de x de
                l'image sur laquelle est appliqué le filtre
                if( x < 0 || x >= imgMaxX){
                    //on est en dehors de l'image
                    continue filtreX;
                }
                filtreY:for(fltY = 0; fltY<fltMaxY; fltY++){
                    y = imgY + fltY - fltOffsetY; //on calcule la valeur de y de
                    l'image sur laquelle est appliqué le filtre
                    if( y < 0 || y >= imgMaxY){
                        //on est en dehors de l'image
                        continue filtreY;
                    }
                }
                //on effectue les sommes
                sommeRouge += image[x][y][0] * filtre[fltX][fltY];
                sommeVert += image[x][y][1] * filtre[fltX][fltY];
            }
        }
    }

```

```

        sommeBleu += image[x][y][2] * filtre[fltX][fltY];
    }
}
//on affecte le résultat au pixel cible
imageFinale[imgX][imgY] = [sommeRouge, sommeVert, sommeBleu];
}
}

return imageFinale;
}

//permet d'afficher le résultat du filtre
function finalisationFiltre(image2D,uid){
    //création du canvas résultat
    var canvasResult = document.createElement("canvas");
    canvasResult.width = canvas.width;
    canvasResult.height = canvas.height;
    canvasResult.className = "filteredCanvas";

    var elem=document.getElementById("filtre_"+uid);
    elem.removeChild(elem.firstChild);
    elem.appendChild(canvasResult);

    elem.parentNode.onmouseover = zoomCanvasOver;
    elem.parentNode.onmouseout = zoomCanvasOut;

    var ctx = canvasResult.getContext("2d");
    var imageData = ctx.getImageData(0, 0, canvasResult.width,
    canvasResult.height);

    var x,
    y,
    lx = image2D.length,
    ly = image2D[0].length,
    i=0,
    image1D = imageData.data;

    for(y = 0; y<ly; y++){
        for(x = 0; x<lx; x++){
            image1D[i++] = image2D[x][y][0];
            image1D[i++] = image2D[x][y][1];
            image1D[i++] = image2D[x][y][2];
            image1D[i++] = image2D[x][y][3] || 255;
        }
    }

    ctx.putImageData(imageData,0,0);
}

//permet d'afficher en plus grand un résultat
function zoomCanvasOver(event){
    var fcanvas = this.lastChild.firstChild;
    if(fcanvas.nodeName !== "CANVAS"){
        return false;
    }
    var ctx = fcanvas.getContext("2d"),
    imageData = ctx.getImageData(0,0,canvas.width,canvas.height);
    ctx = elem_canvasZoom.getContext("2d");
    ctx.putImageData(imageData,0,0);
    elem_zoom.style.display = "block";
}

//enlève l'affichage de résultat
function zoomCanvasOut(event){
    elem_zoom.style.display = "none";
}

/*
 * Initialisation
 */

var uid = 0; //id unique de traitement pour identifier la bonne
ligne
generateFilterList(); //on remplit le select

```

Essayer cet application sans les workers !

Remarque sur l'exemple en ligne : Pour faciliter le test, j'ai ajouté un script qui permet de charger un dessin pré-enregistré.

Solution avec worker

Je vais encore me répéter mais il ne s'agit que d'UNE solution, et il est tout à fait possible de résoudre ce TP d'une manière totalement différente.

Secret (cliquez pour afficher)

Code : HTML - index.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8"/>
  <title>Traitement d'image avec Worker</title>
  <style>
    canvas{
      border: 1px solid #000000;
      z-index:1;
      margin : 0px;
    }
    menu{
      height: 70px;
      margin : 0px;
      padding : 10px;
      margin-right : 300px;
    }
    .filteredCanvas{
      width : 50px;
      height : 25px;
    }
    .zoom{
      position : fixed;
      top : 98px;
      bottom : 10px;
      left : 8px;
      right : 300px;
      display : none;
    }
    .traitement{
      width:300px;
      position:absolute;
      right:0px;
      top:10px;
    }
  </style>
</head>
<body>
  <script src="./canvas.js"></script>
</body>
</html>
```

Code : JavaScript - canvas.js

```
/*
 * création de la zone d'interaction
 */
var canvas_width = document.body.clientWidth - 300;
var canvas_height = Math.round(canvas_width/3);
//alert(canvas_height);

//création d'une zone d'affichage pour les options liées au Canvas
var outils = document.createElement("menu");
document.body.appendChild(outils);

//création du canvas
var canvas = document.createElement("canvas");
canvas.addEventListener("mousedown",function(event){
  canvas.addEventListener("mousemove",draw,false);
  draw(event);
},false);
canvas.addEventListener("mouseup",function(event){
  canvas.removeEventListener("mousemove",draw,false);
},false);
canvas.addEventListener("mouseout",function(event){
  canvas.removeEventListener("mousemove",draw,false);
},false);
canvas.width = canvas_width;
canvas.height = canvas_height;
document.body.appendChild(canvas);

//création des options de modification du pinceau
var elem_couleur = document.createElement("input");
elem_couleur.type = "color";
elem_couleur.value = "#000000";
```

```

elem_couleur.addEventListener("change",function(){ctx.fillStyle=this.value;})
outils.appendChild(elem_couleur);

var elem_taille = document.createElement("input");
elem_taille.type = "number";
elem_taille.min = 1;
elem_taille.value = 10;
elem_taille.title = "Taille du pinceau";
outils.appendChild(elem_taille);

//création d'un bouton de reset
var elem_reset = document.createElement("button");
elem_reset.textContent = "Reset";
elem_reset.addEventListener("click",erase,false);
outils.appendChild(elem_reset);

//Création d'une zone pour réaliser des traitements sur l'image
var elem_zoneTraitement = document.createElement("aside");
elem_zoneTraitement.className = "traitement";
document.body.appendChild(elem_zoneTraitement);

//ajout d'une liste de filtre
elem_zoneTraitement.appendChild( document.createTextNode("Filtre : ") );
var elem_listeFiltre = document.createElement("select");
elem_zoneTraitement.appendChild(elem_listeFiltre);

//ajout du bouton de démarrage
var btn_run = document.createElement("button");
btn_run.textContent = "Appliquer ce filtre";
btn_run.addEventListener("click",prepareFiltre,false);
elem_zoneTraitement.appendChild(btn_run);

//création de la zone de résultat
var elem_result = document.createElement("table");
elem_zoneTraitement.appendChild(elem_result);

//création d'une zone d'affichage pour le zoom du résultat
var elem_zoom = document.createElement("div");
elem_zoom.className = "zoom";
elem_zoneTraitement.appendChild(elem_zoom);

//création d'un canvas pour voir le résultat des filtres
var elem_canvasZoom = document.createElement("canvas");
elem_canvasZoom.width = canvas_width;
elem_canvasZoom.height = canvas_height;
elem_zoom.appendChild(elem_canvasZoom);

/**
 * Fonctions de gestion du Canvas
 */
function erase(){
    ctx.save();
    ctx.fillStyle="#FFFFFF";
    ctx.fillRect(0, 0, canvas.width, canvas.height);
    ctx.restore();
}

//permet de dessiner dans le canvas au niveau du curseur
function draw(event){
    var x = offset_X(event),
        y = offset_Y(event);
    ctx.beginPath();
    ctx.arc(x,y,elem_taille.value,0,Math.PI*2,true);
    ctx.fill();
}

/**
 * Fonctions de gestion des filtres
 */

//ajout d'un script permettant de définir la liste des filtres (création de
listeFiltre)
var scr_filtre = document.createElement("script");
scr_filtre.src = "filtres.js";
document.body.appendChild(scr_filtre);

//permet de préparer la liste des filtres disponibles
function generateFilterList(){
    var i = 0,
        li = listeFiltre.length,
        option;
    do{
        option = document.createElement("option");
        option.textContent = listeFiltre[i].nom;
        option.value = i;
        elem_listeFiltre.add(option, null);
    }while(++i<li);
}

```

```

}

//permet de préparer tout ce qui est nécessaire pour appliquer le filtre sur
function prepareFiltre(){
    var idFiltre = elem_listeFiltre.value;

    var image1D = []; //contiendra la liste des pixels correspondant à l'image
    var imageData = ctx.getImageData(0, 0, canvas.width, canvas.height); //récupère les données binaires du Canvas
    if(imageData){
        // on a réussi à extraire les données du Canvas, dans ce cas on remplit image1D
        image1D = imageData.data;
    } //si on n'a pas réussi à extraire les données, alors on laisse le tableau vide

    //génération d'une nouvelle ligne de résultat
    var ligne = elem_result.insertRow(-1);
    var cellule = ligne.insertCell(0);
    cellule.textContent = listeFiltre[idFiltre].nom;

    cellule = ligne.insertCell(1);
    cellule.id = "filtre_"+uid;

    var progressBar = document.createElement("progress");
    progressBar.max = 200;
    progressBar.id = "progress_"+uid;
    cellule.appendChild(progressBar);

    if(worker){
        //if(image1D instanceof Array){
        try{
            worker.postMessage({idFiltre:idFiltre, image:image1D, width:canvas.width, uid:uid});
        } catch(e) {}
    }
    worker.postMessage({idFiltre:idFiltre, image:JSON.stringify(image1D), width:canvas.width, uid:uid});
    uid++;
    worker = createNewWorker(); //on crée un nouveau worker afin de pouvoir travailler en même temps.
} else{
    var image2D = conversionImage(image1D, canvas.width); //prepare l'image en 2D
    image2D = appliquerFiltre(image2D, idFiltre, uid); //on applique le filtre
    finalisationFiltre(image2D, uid++); //on affiche le résultat
}
}

//permet d'afficher le résultat du filtre
function finalisationFiltre(image2D, uid){
    //création du canvas résultat
    var canvas = document.createElement("canvas");
    canvas.width = canvas_width;
    canvas.height = canvas_height;
    canvas.className = "filteredCanvas";

    var elem=document.getElementById("filtre_"+uid);
    elem.removeChild(elem.firstChild);
    elem.appendChild(canvas);

    elem.parentNode.onmouseover = zoomCanvasOver;
    elem.parentNode.onmouseout = zoomCanvasOut;

    var ctx = canvas.getContext("2d");
    var imageData = ctx.getImageData(0, 0, canvas.width, canvas.height);

    var x,
        y,
        lx = image2D.length,
        ly = image2D[0].length,
        i=0,
        image1D = imageData.data;

    for(y = 0; y<ly; y++){
        for(x = 0; x<lx; x++){
            image1D[i++] = image2D[x][y][0];
            image1D[i++] = image2D[x][y][1];
            image1D[i++] = image2D[x][y][2];
            image1D[i++] = image2D[x][y][3] || 255;
        }
    }

    ctx.putImageData(imageData, 0, 0);
}

//permet d'afficher en plus grand un résultat
function zoomCanvasOver(event){
    var fcanvas = this.lastChild.firstChild;
    if(fcanvas.nodeName !== "CANVAS"){
        return false;
    }
}

```



```

var ctx = fcanvas.getContext("2d"),
    imageData = ctx.getImageData(0,0,canvas_width,canvas_height);
ctx = elem_canvasZoom.getContext("2d");
ctx.putImageData(imageData,0,0);
elem_zoom.style.display = "block";
}

//enlève l'affichage de résultat
function zoomCanvasOut(event){
    elem_zoom.style.display = "none";
}

/**
 * gestion des workers
 */
function createNewWorker(){
    var w = new Worker("traitement.js");
    w.onmessage = function(event){
        var data = event.data;
        switch(data.status){
            case "start": //surtout pour le debug
                console.log("traitement commencé pour "+data.uid);
                break;
            case "debug": //debug
                console.log("debug...");
                console.debug(data.debug);
                break;
            case "update":
                var elem = document.getElementById("progress_"+data.uid);
                if(elem){
                    elem.value = data.progression;
                    elem.textContent = Math.round(data.progression/2) + "%";
                }
                break;
            case "end":
                finalisationFiltre(data.image,data.uid);
                w.terminate(); //le worker a fini sa tâche, on ne le réutilisera plus
                break;
        }
    }
    w.onerror=function(e){console.error(e);}; // pour aider à déboguer
    return w;
}

/**
 * Shim & polyfill
 */
//permet de redéfinir la fonction event.offsetX
if(typeof(offset_X)===undefined){
    function offset_X(event){
        if(event.offsetX) return event.offsetX;
        var el = event.target, ox = -el.offsetLeft;
        while(el=el.offsetParent){
            ox += el.scrollLeft - el.offsetLeft;
        }
        if(window.scrollX){
            ox += window.scrollX;
        }
        return event.clientX + ox -7;
    }
}

//permet de redéfinir la fonction event.offsetY
if(typeof(offset_Y)===undefined){
    function offset_Y(event){
        if(event.offsetY) return event.offsetY;
        var el = event.target, oy = -el.offsetTop;
        while(el=el.offsetParent){
            oy += el.scrollTop - el.offsetTop;
        }
        if(window.scrollY){
            oy += window.scrollY;
        }
        return event.clientY + oy -7;
    }
}

/**
 * Initialisation
 */
var ctx = canvas.getContext("2d"); //référence au contexte de dessin
ctx.fillStyle = elem_couleur.value;

var uid = 0; //id unique pour identifier le traitement

if(window.Worker){
    var worker = createNewWorker();
}else{

```

```

var script_traitement = document.createElement("script");
script_traitement.src = "traitement.js";
document.body.appendChild(script_traitement);
}

window.onload=function() {
  erase();
  generateFilterList();
}

```

Code : JavaScript - filtres.js

```

var listeFiltre = [
  {
    nom:"Flou (petit)",
    filtre:[
      [1/10,1/10,1/10],
      [1/10,2/10,1/10],
      [1/10,1/10,1/10]
    ]
  },
  {
    nom:"Flou (moyen)",
    filtre:[
      [1/26,1/26,1/26,1/26,1/26],
      [1/26,1/26,1/26,1/26,1/26],
      [1/26,1/26,2/26,1/26,1/26],
      [1/26,1/26,1/26,1/26,1/26],
      [1/26,1/26,1/26,1/26,1/26]
    ]
  },
  {
    nom:"Flou (grand)",
    filtre:[
      [1/50,1/50,1/50,1/50,1/50,1/50,1/50],
      [1/50,1/50,1/50,1/50,1/50,1/50,1/50],
      [1/50,1/50,1/50,1/50,1/50,1/50,1/50],
      [1/50,1/50,1/50,2/50,1/50,1/50,1/50],
      [1/50,1/50,1/50,1/50,1/50,1/50,1/50],
      [1/50,1/50,1/50,1/50,1/50,1/50,1/50],
      [1/50,1/50,1/50,1/50,1/50,1/50,1/50]
    ]
  },
  {
    nom:"Flou Gaussien (moyen, σ=0.7)",
    filtre:[
      [0.0001,0.002,0.0055,0.002,0.0001],
      [0.002,0.0422,0.1171,0.0422,0.002],
      [0.0055,0.1171,0.3248,0.1171,0.0055],
      [0.002,0.0422,0.1171,0.0422,0.002],
      [0.0001,0.002,0.0055,0.002,0.0001]
    ]
  },
  /* {
    nom:"Flou Gaussien (grand)",
    filtre:[
      [1/50,1/50,1/50,1/50,1/50,1/50,1/50],
      [1/50,1/50,1/50,1/50,1/50,1/50,1/50],
      [1/50,1/50,1/50,1/50,1/50,1/50,1/50],
      [1/50,1/50,1/50,2/50,1/50,1/50,1/50],
      [1/50,1/50,1/50,1/50,1/50,1/50,1/50],
      [1/50,1/50,1/50,1/50,1/50,1/50,1/50],
      [1/50,1/50,1/50,1/50,1/50,1/50,1/50]
    ]
  }, */
  {
    nom:"Filtre de Laplace (petit)",
    filtre:[
      [-1,-1,-1],
      [-1,8,-1],
      [-1,-1,-1]
    ]
  },
  /* {
    nom:"Filtre de Laplace (moyen)",
    filtre:[
      [1/26,1/26,1/26,1/26,1/26],
      [1/26,1/26,1/26,1/26,1/26],
      [1/26,1/26,2/26,1/26,1/26],
      [1/26,1/26,1/26,1/26,1/26],
      [1/26,1/26,1/26,1/26,1/26]
    ]
  },
  {

```

```

    nom:"Filtre de Laplace (grand)",
    filtre:[
    [1/50,1/50,1/50,1/50,1/50,1/50,1/50],
    [1/50,1/50,1/50,1/50,1/50,1/50,1/50],
    [1/50,1/50,1/50,1/50,1/50,1/50,1/50],
    [1/50,1/50,1/50,2/50,1/50,1/50,1/50],
    [1/50,1/50,1/50,1/50,1/50,1/50,1/50],
    [1/50,1/50,1/50,1/50,1/50,1/50,1/50],
    [1/50,1/50,1/50,1/50,1/50,1/50,1/50]
    ],*/
    {
        nom:"Sobel (vertical)",
        filtre:[
            [-1,0,1],
            [-2,0,2],
            [-1,0,1]
        ]
    },
    {
        nom:"Sobel (horizontal)",
        filtre:[
            [-1,-2,-1],
            [0,0,0],
            [1,2,1]
        ]
    }
    ]
};

```

Code : JavaScript - traitement.js

```

//ajout d'un script permettant de définir la liste des filtres (création de
la variable globale listeFiltre)
if(self.importScripts){
    importScripts("filtres.js");
}

function gestionMessage(event) {
    var data = event.data,
        uid = data.uid;
    self.postMessage({status:"start",uid:uid});
    var image1D = data.image;
    if(typeof image1D === "string"){
        image1D = JSON.parse(image1D);
    }
    self.postMessage({status:"update",uid:uid,progression:0});
    var image2D = conversionImage(image1D,data.width, uid); //prepare l'image
    en 2D (+RVB)
    self.postMessage({status:"update",uid:uid,progression:100});
    image2D = appliquerFiltre(image2D, data.idFiltre, uid); //on applique le
    filtre à l'image
    self.postMessage({status:"end",uid:uid,image:image2D}); //on envoie le
    résultat
}

self.onmessage = gestionMessage;

//permet de convertir un tableaude pixel 1D en 2D
function conversionImage(image1D,w,uid) {
    //prepare l'image en 2D (+RVB)
    var image2D = [],
        i,
        x=0,
        y=0,
        li = image1D.length;

    for(i=0 ; i<li; i++){
        if(y===0){
            image2D[x]=[];
        }
        image2D[x][y]=[];
        image2D[x][y][0]=image1D[i++];
        image2D[x][y][1]=image1D[i++];
        image2D[x][y][2]=image1D[i++];
        if(++x>=w){
            x=0;
            y++;
            if(!(y%30) && typeof window === "undefined"){ //dans le cas où on est
            dans un worker on envoie une mise à jour
                self.postMessage({status:"update",uid:uid,progression:i*100/li});
            }
        }
    }
    return image2D;
}

```

```

//permet d'appliquer le filtre sur l'image
function appliquerFiltre(image, idFiltre, uid){
  var filtre = (listeFiltre[idFiltre] && listeFiltre[idFiltre].filtre) ||
  [], //récupère le filtre s'il existe ou alors génère un filtre vide
  imgX, //position X sur l'image
  imgY, //position Y sur l'image
  imgMaxX = image.length, // largeur de l'image
  imgMaxY = image[0].length, //hauteur de l'image
  fltX, //position X sur le filtre
  fltY, //position Y sur le filtre
  fltMaxX = filtre.length, //largeur du filtre
  fltMaxY = filtre[0].length, //hauteur du filtre
  fltOffsetX = (fltMaxX - 1)/2, //offset X à appliquer sur le filtre pour
  trouver le bon pixel sur l'image
  fltOffsetY = (fltMaxY - 1)/2, //offset Y à appliquer sur le filtre pour
  trouver le bon pixel sur l'image
  index, //sert à identifier la position par rapport à la liste des pixels
  sommeRouge, //valeur temporaire pour l'application du filtre sur un pixel
  sommeVert, //valeur temporaire pour l'application du filtre sur un pixel
  sommeBleu, //valeur temporaire pour l'application du filtre sur un pixel
  x, //index X temporaire pour chercher le bon pixel dans l'image
  y, //index Y temporaire pour chercher le bon pixel dans l'image
  imageFinale=[]; // liste des pixels finales

  //on parcourt tous les pixels de l'image
  imageX:for(imgX = 0; imgX<imgMaxX; imgX++){
    if(!(imgX%20) && typeof window === "undefined"){ //dans le cas où on est
    dans un worker on envoit une mise à jour

self.postMessage({status:"update",uid:uid,progression:100+imgX*100/imgMaxX})
    }
    imageFinale[imgX] = [];
    imageY:for(imgY = 0; imgY<imgMaxY; imgY++){
      sommeRouge = 0;
      sommeVert = 0;
      sommeBleu = 0;

      //on parcourt toutes les valeurs du filtre
      filtreX:for(fltX = 0; fltX<fltMaxX; fltX++){
        x = imgX + fltX - fltOffsetX; //on calcule la valeur de x de l'image sur
        laquelle est appliqué le filtre
        if( x < 0 || x >= imgMaxX){
          //on est en dehors de l'image
          continue filtreX;
        }
        filtreY:for(fltY = 0; fltY<fltMaxY; fltY++){
          y = imgY + fltY - fltOffsetY; //on calcule la valeur de y de l'image
          sur laquelle est appliqué le filtre
          if( y < 0 || y >= imgMaxY){
            //on est en dehors de l'image
            continue filtreY;
          }
          //on effectue les sommes
          sommeRouge += image[x][y][0] * filtre[fltX][fltY];
          sommeVert += image[x][y][1] * filtre[fltX][fltY];
          sommeBleu += image[x][y][2] * filtre[fltX][fltY];
        }
      }
      //on affecte le résultat au pixel cible
      imageFinale[imgX][imgY] = [sommeRouge, sommeVert, sommeBleu];
    }
  }
  return imageFinale;
}

```

[Essayer cette solution !](#)

Remarque sur l'exemple en ligne : Pour faciliter le test, j'ai ajouté un script qui permet de charger un dessin pré-enregistré.

Améliorations possibles

Voici quelques idées pour améliorer cet exercice :

- Permettre de récupérer un résultat pour lui appliquer un filtre.
- Permettre d'effectuer d'autres types de filtres (désaturation, déformation de l'image, etc...).
- Afficher chaque image dans une page distincte. Ainsi une page servira à la gestion des traitements, les autres à l'affichage. Les shared-workers vont vous être utiles pour gérer les modifications effectuées sur une image et les transmettre à la page de gestion. L'utilisateur dispose alors de toute sa page pour modifier/contempler son image. Il peut également ouvrir plusieurs instances de son navigateur pour comparer ses images (à la manière de "The GIMP").

Voilà qui conclut ce tutoriel.

J'espère que cela vous a aidés à comprendre comment fonctionnent les web-workers et vous a donné des idées sur leurs utilisations possibles.

Et voilà vous savez tout sur les web-workers.

Vous allez pouvoir optimiser vos applications javascript et ne pas laisser vos visiteurs poireauter devant un écran inactif. 😎

Maintenant, vous disposez d'un nouvel arsenal pour présenter vos applications web. À vous de savoir en profiter.

Sources et documentations

- [Spécification : Web-workers \(en\)](#)
- [Spécification : Communications \(en\)](#)
- [Documentation de Mozilla : Worker \(en\)](#)
- [Article de Microsoft à propos des web-Workers \(fr\)](#)

Merci à [Golmote](#), [xaviern02](#), et [Nesquick69](#) pour leurs retours et leurs conseils pour améliorer ce tutoriel.

Et un grand merci à [Milena2](#) pour son soutien et ses relectures.