7

Multimédia HTML5 et Web Workers

Nous avons découvert de nombreux moyens de manipuler et de modifier les éléments multimédias HTML5 en JavaScript. Certaines des manipulations vidéo (notamment celles utilisées dans le canevas) peuvent être lentes et gourmandes en ressources processeur. Les Web Workers offrent une solution à ce problème.

Les Web Workers font partie des nouvelles fonctionnalités du HTML5[1]. Ils fournissent une API JavaScript pour exécuter des scripts à l'arrière-plan, indépendamment de tous les scripts d'interface utilisateur, autrement dit parallèlement à la page principale, sans en perturber le fonctionnement. N'importe quel programme JavaScript peut être converti en un Web Worker. Pour l'interaction avec la page principale, vous devez utiliser le passage de message, car les Web Workers n'ont pas accès au DOM de la page web. Les Web Workers introduisent en quelque sorte le multithreading sur le Web.

Nous ne proposerons pas ici une étude approfondie des Web Workers. D'autres livres ou articles HTML5 s'en chargent déjà. Mozilla[2] en donne une bonne introduction par John Resig[3] à l'adresse **https://developer.mozilla.org/En/ Using_web_workersor**. Pour notre part, nous verrons plus spécifiquement comment utiliser les Web Workers avec les éléments multimédias HTML5. Comme chaque technologie du HTML5, les Web Workers sont encore nouveaux et l'on peut s'attendre à ce que leurs spécifications et leurs implémentations s'améliorent. Voici les fonctions et les événements que les Web Workers assurent pour le moment :

- le constructeur `Worker()`, qui définit un fichier JavaScript en qualité de Web Worker ;
- les événements `message` et `error` pour le worker ;

1. Voir **http://www.whatwg.org/specs/web-workers/current-work/**.
2. Voir **https://developer.mozilla.org/En/Using_web_workers**.
3. Voir **http://ejohn.org/blog/web-workers/**.

© 2012 Pearson France – HTML5 <video> – Silvia Pfeiffer

- la méthode `postMessage()` utilisée par un Web Worker et qui active le gestionnaire d'événements `message` de la page principale ;
- la méthode `postMessage()` utilisée par la page principale qui active le gestionnaire d'événements `message` du Web Worker afin de répondre ;
- JSON, utilisé pour le passage des messages ;
- la méthode `terminate()` utilisée par la page principale pour terminer immédiatement le Web Worker ;
- l'événement `error` constitué d'un champ de message lisible par l'homme, du nom de fichier (`filename`) du Web Worker et du numéro de ligne (`lineno`) où l'erreur s'est produite ;
- la méthode `importScripts()` utilisée par le Web Worker pour charger des fichiers JavaScript partagés ;
- la possibilité pour les Web Workers d'invoquer `XMLHttpRequest`.

Les Web Workers ne fonctionnent que lorsqu'ils sont utilisés sur un serveur web, car le script externe doit être chargé avec le même schéma que la page d'origine. Cela signifie que vous ne pouvez pas charger un script depuis une URL `data:`, `javascript:` ou `file:`. En outre, une page `https:` ne peut lancer que des Web Workers qui se trouvent également sur `https:` et non sur `http:`.

Notez par ailleurs que la version d'Internet Explorer utilisée pour ce livre ne les prend pas encore en charge. À cause du bogue `getImageData()` et `putImageData()` mentionné au chapitre précédent, nous n'avons pu faire fonctionner aucun des exemples sans Web Worker de ce chapitre dans Internet Explorer. Nous ne proposerons donc pas de captures d'écran pour ce navigateur.

Utilisation de Web Workers avec la vidéo

Au long de cette section, nous allons voir comment transformer une page vidéo HTML5 qui manipule des données vidéo en JavaScript de manière à exécuter ces opérations séparément, dans un thread de Worker, avant de les restituer ultérieurement à la page web principale.

Pour cet exemple, nous utilisons un duplicata en tons sépia de la vidéo dans le canevas. Le Listing 7.1 montre comment procéder quand aucun Web Worker n'est utilisé.

© 2012 Pearson France – HTML5 <video> – Silvia Pfeiffer

Listing 7.1 : Coloriage sépia des pixels vidéo dans le canevas

```
<video controls height="270px" width="480px" >
    <source src="HelloWorld.mp4" type="video/mp4">
    <source src="HelloWorld.webm" type="video/webm">
    <source src="HelloWorld.ogv" type="video/ogg">
</video>
<canvas width="400" height="300" style="border: 1px solid black;"> </canvas>
<canvas id="scratch" width="400" height="300" style="display: none;"> </canvas>
<script>
    window.onload = function() { initCanvas(); }
    var context, video, sctxt;
    function initCanvas() {
            video = document.getElementsByTagName("video")[0];
            canvas = document.getElementsByTagName("canvas")[0];
            context = canvas.getContext("2d");
            scratch = document.getElementById("scratch");
            sctxt = scratch.getContext("2d"); video.addEventListener("play",
playFrame, false);
    }
    function playFrame() {
            w = 320;
            h = 160;
            sctxt.drawImage(video, 0, 0, w, h);
            frame = sctxt.getImageData(0, 0, w, h);
            // Parcourir en boucle chaque pixel de l'image
            for (x = 0; x < w; x ++) {
                    for (y = 0; y < h; y ++) {
                    // Index dans le tableau de données d'image
                    i = x + w*y;
                    // Récupérer les couleurs
                    r = frame.data[4*i+0];
                    g = frame.data[4*i+1];
                    b = frame.data[4*i+2];
                    // Remplacer par les couleurs sépia
                    frame.data[4*i+0] = Math.min(0.393*r + 0.769*g +
0.180*b,255);
                    frame.data[4*i+1] = Math.min(0.349*r + 0.686*g +
0.168*b,255);
                    frame.data[4*i+2] = Math.min(0.272*r + 0.534*g +
0.131*b,255);
                    }
            }
            context.putImageData(frame, 0, 0);
            if (video.paused || video.ended) { return; }
            setTimeout(function () { playFrame(); }, 0);
    }
</script>
```

© 2012 Pearson France – HTML5 <video> – Silvia Pfeiffer

Chaque pixel de l'image qui est importé dans le canevas de travail est remplacé par une nouvelle valeur RVB calculée sous forme de mélange sépia des couleurs existantes[1]. L'image modifiée est ensuite écrite dans un canevas visible.

Notez qu'il faut veiller à ce que les nouvelles valeurs de couleur ne dépassent pas 255, car seuls 8 bits sont utilisés pour stocker les couleurs. Toute valeur supérieure à 255 peut générer un dépassement de valeur et donc une couleur inexacte. Cela se produit de fait dans Opera, les autres navigateurs limitant la valeur avant de l'attribuer. Dans tous les cas, la méthode la plus sûre consiste à utiliser une fonction `Math.min` sur les valeurs de couleur.

La Figure 7.1 présente le résultat. Si vous examinez l'exemple en couleur, vous verrez que la vidéo du haut est en couleur et que le canevas apparaît en tons sépia.

Figure 7.1
Peinture d'un duplicata vidéo en tons sépia dans un canevas sous Firefox, Safari, Opera et Chrome (de gauche à droite).

Tentons maintenant d'accélérer le calcul des couleurs sépia (le programme parcourt en boucle chaque pixel et chaque composante de couleur des images vidéo capturées) en déléguant les tâches de calcul JavaScript lourdes à un Web Worker. Nous opérerons une comparaison des vitesses par la suite. Le Listing 7.2 présente le code de la page web et le Listing 7.3 le code JavaScript correspondant au Web Worker du Listing 7.2. Le code Web Worker est situé dans une autre ressource JavaScript appelée worker.js. Elle doit être fournie à partir du même domaine que la page web principale. Il s'agit actuellement du seul moyen d'appeler un Web Worker. Des discussions ont été entamées afin d'étendre cette fonctionnalité et d'autoriser la définition incorporée des Web Workers[2].

1. Selon la recette de mélange publiée à l'adresse **http://www.builderau.com.au/program/csharp/soa/How-do-I-convert-images-to-grayscale-and-sepia-tone-using-C-/0,339028385,339291920,00.htm**.
2. Voir **http://lists.whatwg.org/pipermail/whatwg-whatwg.org/2010-October/028844.html**.

© 2012 Pearson France – HTML5 <video> – Silvia Pfeiffer

Listing 7.2 : Coloriage sépia des pixels vidéo à l'aide d'un Web Worker

```
<video controls height="270px" width="480px" >
    <source src="HelloWorld.mp4" type="video/mp4">
    <source src="HelloWorld.webm" type="video/webm">
    <source src="HelloWorld.ogv" type="video/ogg">
</video>
<canvas width="400" height="300" style="border: 1px solid black;"> </canvas>
<canvas id="scratch" width="400" height="300" style="display: none;"> </canvas>
<script>
    window.onload = function() { initCanvas(); }
    var worker = new Worker("worker.js");
    var context, video, sctxt;
    function initCanvas() {
            video = document.getElementsByTagName("video")[0];
            canvas = document.getElementsByTagName("canvas")[0];
            context = canvas.getContext("2d");
            scratch = document.getElementById("scratch");
            sctxt = scratch.getContext("2d");
            video.addEventListener("play", postFrame, false);
            worker.addEventListener("message", drawFrame, false);
    }
    function postFrame() {
            w = 320;
            h = 160;
            sctxt.drawImage(video, 0, 0, w, h);
            frame = sctxt.getImageData(0, 0, w, h);
            arg = { frame: frame, height: h, width: w }
            worker.postMessage(arg);
    }
    function drawFrame (event) {
            outframe = event.data;
            if (video.paused || video.ended) { return; }
            context.putImageData(outframe, 0, 0);
            setTimeout(function () { postFrame(); }, 0);
    }
</script>
```

Dans le Listing 7.2, nous avons marqué les nouvelles commandes en gras. Observez de quelle manière le Web Worker est créé. Un message est préparé et transmis. Une fonction récupère sous forme de message l'image colorée en tons sépia quand le Web Worker a terminé, puis l'envoie. La clé est ici d'avoir séparé la préparation des données et les calculs dans `postFrame()` et le dessin des résultats dans `drawFrame()`.

© 2012 Pearson France – HTML5 <video> – Silvia Pfeiffer

Le Web Worker qui réalise les calculs est stocké dans un fichier distinct, ici appelé worker.js. Il ne contient que le rappel pour l'événement `onmessage` de la page web. Il n'a pas d'autres données ou fonctions à initialiser. Il reçoit l'image originale de la page web, calcule les nouvelles valeurs de pixels, les remplace dans l'image et repasse l'image redessinée à la page web.

Listing 7.3 : Web Worker JavaScript du Listing 7.2

```
onmessage = function (event) {
    // Récupérer les données d'image
    var data = event.data;
    var frame = data.frame;
    var h = data.height;
    var w = data.width;
    var x,y;
    // Parcourir en boucle chaque pixel de l'image
    for (x = 0; x < w; x ++) {
        for (y = 0; y < h; y ++) {
            // Index dans l'image
            i = x + w*y;
            // Récupérer les couleurs
            r = frame.data[4*i+0];
            g = frame.data[4*i+1];
            b = frame.data[4*i+2];
            // Remplacer par les couleurs sépia
            frame.data[4*i+0] = Math.min(0.393*r + 0.769*g + 0.189*b, 255);
            frame.data[4*i+1] = Math.min(0.349*r + 0.686*g + 0.168*b, 255);
            frame.data[4*i+2] = Math.min(0.272*r + 0.534*g + 0.131*b, 255);
            }
    }
    // Retransmettre les données d'image au thread principal
    postMessage(frame);
}
```

Nous avons là un bon exemple de manipulation d'une vidéo avec un Web Worker. On ne peut pas transmettre un canevas directement à un Web Worker, par exemple comme paramètre de la fonction `postMessage()`, car il s'agit d'un élément du DOM, alors que les Web Workers ne reconnaissent pas les éléments du DOM. En revanche, vous pouvez passer les données d'image au Worker. La méthode pour manipuler une vidéo consiste donc à récupérer une image vidéo avec `getImageData()`, à la placer dans un message et à transmettre ce message au Web Worker avec `postMessage()`, où l'événement `message` déclenche l'exécution de l'algorithme de manipulation de la vidéo. Le résultat des calculs est renvoyé au thread principal par le Web Worker *via* un appel `postMessage()` avec les données d'image manipulées en paramètre. Le contrôle est ainsi redonné au gestionnaire

© 2012 Pearson France – HTML5 <video> – Silvia Pfeiffer

d'événements `onmessage` du thread principal afin d'afficher l'image manipulée en utilisant `putImageData()` dans le canevas.

Les Web Workers sont pris en charge dans tous les navigateurs à l'exception d'Internet Explorer. Le résultat de l'implémentation avec Web Workers de l'exemple de conversion en tons sépia est identique à celui de l'implémentation sans Worker (voir Figure 7.1).

Si vous réalisez vos développements sous Opera et que vous comptiez recharger votre Web Worker en effectuant un rechargement de la page web avec la touche Maj enfoncée, vous risquez d'être déçu. Assurez-vous donc de conserver un onglet supplémentaire ouvert avec un lien vers le fichier JavaScript du Web Worker et effectuez séparement le rechargement des deux pages.

L'exemple sépia est simple, si bien qu'on peut légitimement se demander si la surcharge résultant de la préparation du message (copier les données du message, en incluant l'image), son dépaquetage et ces opérations reproduites pour le résultat ajoutées au délai d'appel des événements, ne surpasse pas finalement le gain de temps obtenu en déléguant la manipulation des données vidéo à un thread.

Il faut comparer le nombre d'images manipulées lors de l'exécution dans le thread principal avec le nombre d'images qu'un Web Worker traite pour déterminer les limites de cette approche. Notez que cette méthode est limitée par l'obligation que nous nous imposons de conserver l'affichage du Web Worker et la lecture de la vidéo approximativement synchronisés au lieu d'autoriser le Web Worker à être plus lent. Le Tableau 7.1 présente le résultat avec le nombre d'images traitées pour la vidéo d'exemple Hello Word de quatre secondes.

Tableau 7.1 : Performances des navigateurs sans (à gauche) et avec (à droite) des Web Workers pour l'exemple des tons sépia

Firefox	*Safari*	*Chrome*	*Opera*
89 53 (WW)	87 96 (WW)	77 54 (WW)	93 95 (WW)

Nous avons obtenu les résultats du Tableau 7.1 sur un même ordinateur, avec une charge identique, en rechargeant l'exemple plusieurs fois et en enregistrant le nombre maximal d'images recolorées atteint par chaque navigateur. Notez que l'algorithme avec Web Workers traite moins d'images sur Firefox et Chrome que lorsque le code est exécuté sur la page web principale. Pour Safari, il y a un gain de vitesse, alors qu'Opera présente à peu près les mêmes performances avec ou sans Web Worker.

Ces résultats semblent dépendre de la manière dont les navigateurs implémentent la prise en charge des Web Workers. Notez que nous ne comparons pas les performances entre les différents navigateurs, qui dépendent clairement de leurs moteurs JavaScript, mais on peut voir l'effet combiné de la manière dont chaque navigateur implémente les Web Workers et de la vitesse de son moteur JavaScript.

© 2012 Pearson France – HTML5 <video> – Silvia Pfeiffer

Opera est conçu comme un navigateur à thread unique, aussi son implémentation actuelle de Web Workers intercale-t-elle l'exécution du code dans le thread unique. L'implémentation Mozilla dans Firefox est différente, puisque le Web Worker est un véritable thread de système d'exploitation et que la division des workers permet de tirer parti des multiples cœurs de processeur. La surcharge introduite par le fait de générer un thread de niveau système complet dans notre simple exemple ici semble avoir un impact négatif sur le nombre d'images qui peuvent être décodées et transmises depuis le thread principal durant le temps de lecture.

Le principal avantage de l'utilisation d'un Web Worker tient à ce que la surcharge du thread principal est réduite de telle sorte qu'il peut continuer à s'exécuter à pleine vitesse. En particulier, il peut ainsi continuer d'afficher l'interface utilisateur du navigateur, de vérifier vos e-mails, etc. Dans notre exemple, cela se manifeste particulièrement par la vitesse de lecture de la vidéo. Dans l'exemple de la conversion en tons sépia, notre thread principal n'était pas particulièrement surchargé de calculs, aussi l'introduction des Web Workers n'a-t-elle pas apporté d'amélioration importante. Passons donc maintenant à un exemple plus musclé.

Détection de mouvement avec des Web Workers

L'idée générale de la détection de mouvement consiste à considérer la différence entre deux images successives dans une vidéo et à déterminer s'il existe un changement suffisamment important pour conclure qu'il s'agit d'un mouvement. Les bons détecteurs de mouvement peuvent gérer les changements de conditions lumineuses, les caméras qui se déplacent et les artéfacts de caméra et d'encodage. Pour notre exemple, nous allons simplement déterminer si un pixel a changé afin de savoir s'il y a eu un mouvement. C'est une approche simple mais plutôt efficace qui conviendra pour les besoins de la démonstration.

Niveaux de gris

L'approche pratique de la détection de mouvement implique de prétraiter les images en les passant en niveaux de gris. Comme la couleur n'influence pas le mouvement, c'est une abstraction raisonnable, qui réduit le nombre de calculs nécessaires, puisque les différences n'ont pas besoin d'être calculées sur toutes les couches, mais uniquement sur une seule.

On réalise le passage en niveaux de gris en calculant la luminance (l'intensité lumineuse) de chaque pixel et en remplaçant les valeurs des couches rouge, vert et bleu par cette valeur de luminance. Toutes les couches possédant alors des valeurs identiques, l'image n'a plus de couleur et apparaît en gris.

On peut calculer la luminance à partir d'une moyenne des valeurs originales rouge, vert et bleu, mais cette méthode ne concorde pas avec la perception humaine de la luminance. En réalité, le meilleur moyen de calculer la luminance consiste à

© 2012 Pearson France – HTML5 <video> – Silvia Pfeiffer

prendre 30% de rouge, 59% de vert et 11% de bleu[1]. Le bleu est perçu comme une couleur très sombre et le vert comme une couleur très claire, de sorte que ces deux couleurs contribuent différemment à la perception humaine de la luminance.

Le Listing 7.4 présente le Web Worker JavaScript qui crée une version en niveaux de gris de la vidéo à l'aide de cet algorithme. Le code du thread principal associé à ce Web Worker est identique à celui du Listing 7.2. La Figure 7.2 présente les captures d'écran résultant des différents navigateurs. La vidéo du haut est en couleur et le canevas en dessous est en noir et blanc.

Listing 7.4 : Conversion des pixels vidéo en niveaux de gris à l'aide d'un Web Worker

```
onmessage = function (event) {
    // Récupérer les données d'image
    var data = event.data;
    var frame = data.frame;
    var h = data.height;
    var w = data.width;
    var x,y;
    // Parcourir en boucle chaque pixel de l'image
    for (x = 0; x < w; x ++) {
            for (y = 0; y < h; y ++) {
                    // Index dans le tableau des données d'image
                    i = x + w*y;
                    // Récupérer les couleurs
                    r = frame.data[4*i+0];
                    g = frame.data[4*i+1];
                    b = frame.data[4*i+2];
                    col = Math.min(0.3*r + 0.59*g + 0.11*b, 255);
                    // Remplacer par la teinte noir et blanc
                    frame.data[4*i+0] = col;
                    frame.data[4*i+1] = col;
                    frame.data[4*i+2] = col;
            }
    }
    // Retransmettre les données d'image au thread principal
    postMessage(frame);
}
```

1. Pour plus d'informations concernant la perception de la luminance, consultez la section correspondante de l'article Wikipédia : **http://fr.wikipedia.org/wiki/Luminance#Perception_visuelle_de_la_luminance**.

© 2012 Pearson France – HTML5 <video> – Silvia Pfeiffer

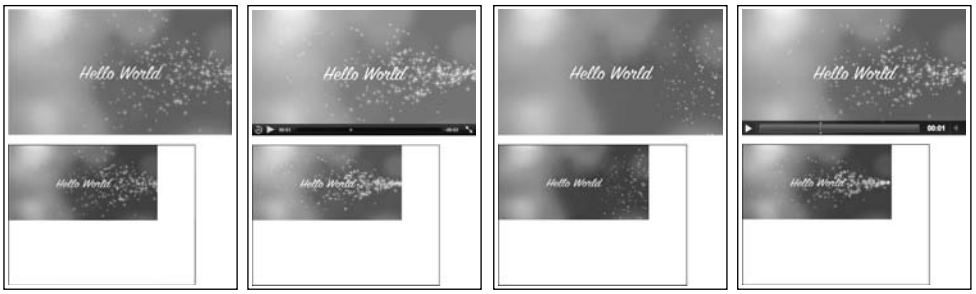

Figure 7.2
Peinture d'un duplicata vidéo en niveaux de gris dans un canevas sous Firefox, Safari, Opera et Chrome (de gauche à droite).

Nous disposons maintenant d'un algorithme capable de créer une image en niveaux de gris, mais il est intéressant de noter qu'il n'est pas nécessaire d'utiliser l'image en niveaux de gris complète pour calculer la différence de mouvement entre deux images de ce type. De nombreuses répétitions peuvent être évitées dans les trois couches de couleur et, en outre, nous n'avons pas à nous occuper de la couche alpha. Nous pouvons donc réduire les images à un simple tableau de valeurs de luminance.

Détection de mouvement

Passons maintenant à l'implémentation de la détection de mouvement. Pour visualiser les pixels qui ont été identifiés comme pixels de mouvement, nous allons les peindre avec une couleur qui se remarque. Nous choisissons un mélange de vert = 100 et bleu = 255. Le Listing 7.5 présente le Web Worker qui implémente la détection de mouvement. Le code du thread principal est le même que celui du Listing 7.2.

Listing 7.5 : Détection de mouvement de pixels vidéo utilisant un Web Worker

```
var prev_frame = null;
var threshold = 25;
function toGray(frame) {
    grayFrame = new Array (frame.data.length / 4);
    for (i = 0; i < grayFrame.length; i++) {
            r = frame.data[4*i+0];
            g = frame.data[4*i+1];
            b = frame.data[4*i+2];
            grayFrame[i] = Math.min(0.3*r + 0.59*g + 0.11*b, 255);
    }
    return grayFrame;
}
onmessage = function (event) {
```

© 2012 Pearson France – HTML5 <video> – Silvia Pfeiffer

```
        // Récupérer les données d'image
        var data = event.data;
        var frame = data.frame;
        // Convertir l'image courante en niveaux de gris
        cur_frame = toGray(frame);
        // Éviter cet appel la première fois
        if (prev_frame != null) {
                // Calculer la différence
                for (i = 0; i < cur_frame.length; i++) {
                        if (Math.abs(prev_frame[i] - cur_frame[i]) > threshold) {
                                // Colorer les pixels très différents
                                frame.data[4*i+0] = 0;
                                frame.data[4*i+1] = 100; frame.data[4*i+2] = 255;
                        }
                }
        }
        // Mémoriser l'image actuelle comme image précédente
        prev_frame = cur_frame;
        // Retransmettre les données d'image au thread principal
        postMessage(frame);
}
```

Vous aurez remarqué que ce Web Worker utilise en fait des données globales car nous devons mémoriser les données de la précédente image entre les différents appels au Web Worker. Nous initialisons ce tableau en lui attribuant la valeur `null` afin d'éviter de réaliser le calcul de différence lors du premier appel au Web Worker. L'autre variable globale est le seuil (`threshold`) que nous avons choisi de fixer à 25, ce qui fournit une tolérance au bruit plutôt raisonnable.

Vous aurez sans doute reconnu la fonction `toGray()` du précédent algorithme. Cette fois, nous ne stockons pour chaque image que le tableau réduit des valeurs de gris.

Dans la fonction de rappel de l'événement `onmessage`, nous commençons par calculer la version en niveaux de gris de l'image courante, puis nous utilisons ces informations pour les comparer à celles de la précédente image et colorons les pixels dont la différence de luminance dépasse le seuil de référence. Nous mémorisons ensuite les valeurs de luminance de l'image actuelle dans la variable `prev_frame` pour l'itération suivante et repostons l'image ajustée à destination du thread principal qui doit l'afficher.

La Figure 7.3 présente le résultat de cet algorithme appliqué à la vidéo "Hello World" dans tous les navigateurs à l'exception d'Internet Explorer.

© 2012 Pearson France – HTML5 <video> – Silvia Pfeiffer

Figure 7.3
Résultats de la détection de mouvement sur une vidéo avec des Web Workers dans Firefox, Safari, Opera et Chrome (de gauche à droite).

Comme la vidéo “Hello World” n’est pas très excitante pour les besoin de démonstration du mécanisme de détection de mouvement, nous avons reproduit les captures résultantes de l’assemblage sur des scènes de la vidéo *Elephants Dream*.

Figure 7.4
Résultats de la détection de mouvement sur une seconde vidéo avec des Web Workers dans Firefox, Safari, Opera et Chrome (de gauche à droite).

Vous remarquerez certainement les défauts de cet algorithme si vous le faites fonctionner avec vos propres vidéos. Vous pouvez rechercher des idées pour en améliorer les performances ou l’adapter à vos besoins, par exemple pour un système d’alerte sur une caméra de surveillance. Il existe bien des algorithmes de meilleure facture pour la détection de mouvement, mais ce n’est pas le sujet de ce livre.

Examinons à nouveau les performances de cet algorithme dans les différents navigateurs. Le Tableau 7.2 présente les différences entre une implémentation avec Web Workers et une autre qui n’y fait pas appel. Le nombre indiqué représente le nombre d’images affichées dans le canevas quand l’algorithme est exécuté avec ou sans Web Workers pour la vidéo “Hello World” de quatre secondes.

© 2012 Pearson France – HTML5 <video> – Silvia Pfeiffer

Tableau 7.2 : Performances des navigateurs sans (à gauche) et avec (à droite) Web Workers pour la détection de mouvement

Firefox	*Safari*	*Chrome*	*Opera*
82 48 (WW)	64 62 (WW)	105 75 (WW)	140 129 (WW)

Dans le cas présent, deux boucles sont impliquées dans chaque itération du Web Worker et des données globales doivent être stockées. L'implémentation sur la page web principale parvient à manipuler plus d'images pour tous les navigateurs. La différence n'est pas importante pour Safari et Opera, mais elle est étonnamment grande pour Firefox et Chrome. Cela signifie que le code du Web Worker est en fait assez lent et ne peut suivre le rythme de lecture de la vidéo. Les Web Workers délivrent donc le thread principal d'une charge importante et permettent ainsi à la vidéo d'être lue avec moins d'effort. Les différences ne sont pas encore visibles quand on exécute l'algorithme avec ou sans Web Workers, car la lecture vidéo s'effectue de manière fluide dans les deux cas. Augmentons donc encore la complexité en présentant une autre opération de traitement vidéo.

Segmentation des zones

Dans le domaine du traitement d'image et *a fortiori* du traitement vidéo, la segmentation de l'image affichée en zones d'intérêt requiert généralement des ressources processeur importantes. La segmentation d'images est utilisée pour localiser des objets et des contours (ligne, courbes, etc.) dans les images en visant à donner la même étiquette aux zones qui appartiennent au même ensemble.

Au sein de cette section, nous allons implémenter une méthode simple de segmentation de zones et montrer comment utiliser des Web Workers pour les tâches intensives et libérer le thread principal afin d'obtenir une lecture vidéo fluide.

Notre segmentation de zones se fonde sur les pixels identifiés par la détection de mouvement à partir de l'algorithme de la précédente section. Par une sorte de méthode de grossissement des zones[1], nous allons regrouper les pixels de mouvement qui ne sont pas trop éloignés les uns des autres. Dans notre exemple, le seuil de distance est fixé à 2. Nous limitons le regroupement à une zone de 5 × 5 pixels autour du pixel de mouvement. Ce regroupement peut amener à fusionner dans une même zone un grand nombre de pixels. Nous affichons un rectangle autour de tous les pixels dans la plus grande zone trouvée dans chaque image.

Commençons par développer une version sans Web Workers. En général, cette méthodologie est préférable, car elle facilite le débogage. Pour l'instant, il n'existe pas de moyen de déboguer facilement un Web Worker dans un navigateur. Pour

1. Voir **http://fr.wikipedia.org/wiki/Segmentation_d%27image**.

© 2012 Pearson France – HTML5 <video> – Silvia Pfeiffer

autant que vous gardiez à l'esprit que vous allez séparer le code JavaScript dans un Web Worker, il est plus simple de commencer par un thread unique.

Le Listing 7.6 présente la fonction `playFrame()` qu'utilise la page web pour la segmentation. Le reste de la page est identique au Listing 7.1. Le code utilise en outre la fonction `toGray()` présentée dans le Listing 7.5. Cette fonction paraît bien longue et intimidante, mais elle est en fait composée de beaux blocs ponctués de commentaires, que nous allons ensuite parcourir pas à pas.

Listing 7.6 : Segmentation de pixels vidéo à l'aide d'un Web Worker

```
// Initialisation pour la segmentation
var prev_frame = null;
var cur_frame = null;
var threshold = 25;
var width = 320;
var height = 160;
region = new Array (width*height);
index = 0;
region[0] = {};
region[0]['weight'] = 0;
region[0]['x1'] = 0;
region[0]['x2'] = 0;
region[0]['y1'] = 0;
region[0]['y2'] = 0;
function playFrame() {
     sctxt.drawImage(video, 0, 0, width, height);
     frame = sctxt.getImageData(0, 0, width, height);
     cur_frame = toGray(frame);
     // Éviter le calcul pour la première image
     if (prev_frame != null) {
     // Initialiser les champs de zones
     for (x = 0; x < width; x++) {
             for (y = 0; y < height; y++) {
                     i = x + width*y;
                     // Initialiser les champs de zones
                     if (i != 0) region[i] = {};
                     region[i]['weight'] = 0;
                     if (Math.abs(prev_frame[i] - cur_frame[i]) > threshold) {
                             // Initialiser les zones
                                     region[i]['weight'] = 1;
                                     region[i]['x1'] = x;
                                     region[i]['x2'] = x;
                                     region[i]['y1'] = y;
                                     region[i]['y2'] = y;
                             }
```

© 2012 Pearson France – HTML5 <video> – Silvia Pfeiffer

```
                        }
                }
                // Segmentation : élargir la zone autour de chaque pixel de mouvement
                for (x = 0; x < width; x++) {
                        for (y = 0; y < height; y++) {
                                i = x + width*y;
                                if (region[i]['weight'] > 0) {
                                        // Vérifier les voisins sur une grille de
5x5
                                        for (xn = Math.max(x-2,0); xn <= Math.
min(x+2,width-1); xn++) {
                                                for (yn = Math.max((y-2),0); yn <=
Math.min((y+2),(height-1)); yn++) {
                                                        j = xn + width*yn;
                                                        if (j != i) {
                                                                if (region[j]
['weight'] > 0) {
                                                                        region[i]
['weight'] += region[j]['weight'];
                                                                        region[i]
['x1'] = Math.min(region[i]['x1'], region[j]['x1']);
                                                                        region[i]
['y1'] = Math.min(region[i]['y1'], region[j]['y1']);
                                                                        region[i]
['x2'] = Math.max(region[i]['x2'], region[j]['x2']);
                                                                        region[i]
['y2'] = Math.max(region[i]['y2'], region[j]['y2']);
                                                                }
                                                        }
                                                }
                                        }
                                }
                        }
                }
                // Trouver l'un des pixels les plus lourds (l'un des groupes les plus
grands)
                max = 0;
                index = 0;
                // Réinitialiser
                for (x = 0; x < width; x++) {
                        for (y = 0; y < height; y++) {
                                i = x + width*y;
                                if (region[i]['weight'] > max) {
                                        max = region[i]['weight'];
                                        index = i;
                                }
```

```
                    }
            }
    }
    // Mémoriser l'image actuelle comme image précédente et récupérer les coor-
données du rectangle
    prev_frame = cur_frame;
    x = region[index]['x1'];
    y = region[index]['y1'];
    w = (region[index]['x2'] - region[index]['x1']);
    h = (region[index]['y2'] - region[index]['y1']);
    // Dessiner l'image et le rectangle
    context.putImageData(frame, 0, 0);
    context.strokeRect(x, y, w, h);
    calls += 1;
    if (video.paused || video.ended) { return; }
    setTimeout(function () { playFrame(); }, 0);
}
```

Le code commence par une initialisation de la structure de mémoire requise pour opérer la segmentation. Les représentations `prev_frame` et `cur_frame` sont des représentations en niveaux de gris de l'image précédente et de l'image actuelle qui se trouvent comparées. Le seuil (`threshold`), comme auparavant, identifie les pixels en mouvement. Les variables `width` (largeur) et `height` (hauteur) identifient les dimensions de l'affichage de la vidéo dans le canevas. Le tableau `region` est un tableau de hachages contenant des informations concernant chaque pixel d'image actuellement considéré : son poids vaut initialement 1, mais il grossit à mesure que d'autres pixels sont repérés dans son voisinage ; les coordonnées (x1,y1) et (x2,y2) désignent la zone à laquelle des poids de pixels ont été ajoutés. La variable `index` correspond à la fin à l'indice du tableau de zone avec le regroupement le plus large.

Dans `playFrame()`, nous commençons par extraire l'image courante de la vidéo et calculons sa représentation en niveaux de gris. Nous ne réalisons la segmentation que s'il ne s'agit pas de la toute première image. S'il s'agit de la première image, une zone de (0,0) à (0,0) serait créée et peinte sur le canevas.

Pour réaliser la segmentation, nous commençons par initialiser les champs de zone. Seuls ceux qui sont considérés comme des pixels de mouvement se voient attribuer le poids (`weight`) 1 et une zone initiale simplement constituée de leur propre pixel.

Ensuite, nous réalisons l'agrandissement de zone sur la grille de 5 × 5 pixels qui entoure ces pixels de mouvement. Nous ajoutons au pixel courant le poids de tous les pixels de mouvement repérés dans la zone qui l'entoure et définissons l'étendue de la zone en la faisant correspondre au rectangle le plus large qui inclut ces autres pixels de mouvement.

Puisque nous ne souhaitons marquer qu'une seule zone, nous identifions ensuite le dernier des plus importants regroupements, autrement dit celui qui a été trouvé

© 2012 Pearson France – HTML5 <video> – Silvia Pfeiffer

autour d'un des pixels les plus lourds (dont le poids est le plus important). C'est ce regroupement que nous allons peindre sous forme de rectangle. Nous attribuons donc à la variable `index` la valeur d'index de ce pixel dans le tableau `region`.

Pour finir, nous pouvons déterminer les coordonnées et peindre l'image et le rectangle dans le canevas. Nous définissons ensuite un temps imparti sur un autre appel à la fonction `playFrame()`, qui permet au thread principal d'exécuter la lecture vidéo avant de réaliser l'analyse d'image à nouveau pour l'image suivante.

Notez que, dans certaines circonstances, cette méthode peut générer des calculs incorrects pour l'étendue de la zone. Chaque fois qu'une forme verticale ou horizontale rétrécit au lieu de continuer de s'agrandir, le dernier pixel de mouvement vérifié possède le plus gros poids, mais n'a pas reçu l'étendue complète de la zone. Un deuxième passage dans cette zone serait nécessaire pour en déterminer la taille effective. Nous laissons le soin au lecteur d'adapter le code pour l'améliorer.

La Figure 7.5 présente le résultat de cet algorithme appliqué à la vidéo "Hello World".

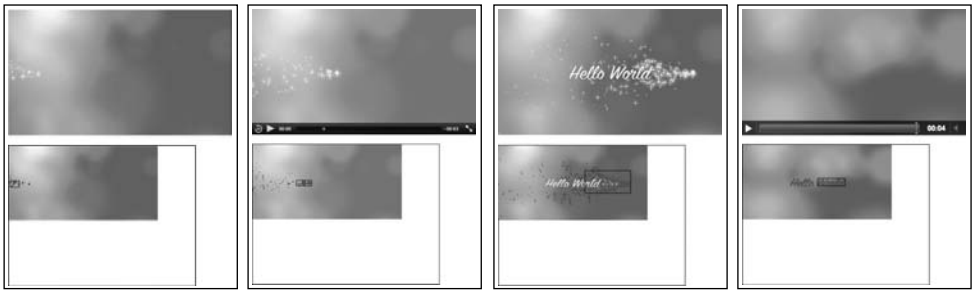

Figure 7.5
Résultats de la segmentation d'image sur une détection de mouvement vidéo dans Firefox, Safari, Opera et Chrome (de gauche à droite).

Vous remarquerez que la lecture vidéo est maintenant sérieusement perturbée dans tous les navigateurs à l'exception de Safari. Les vidéos ont l'air saccadées et il semble évident que le navigateur a du mal à trouver suffisamment de cycles à consacrer au décodage vidéo au lieu de les occuper sur les tâches JavaScript. Voilà un cas idéal pour tirer parti des Web Workers. L'option qui consisterait à réduire la fréquence à laquelle l'analyse est opérée fonctionnerait également, mais elle ne profite pas d'une mise à l'échelle relative aux capacités du navigateur.

Nous avons conçu la base de code de sorte qu'il soit facile de déplacer dans un Web Worker les portions du code relatives à la manipulation. Nous allons transmettre l'image vidéo courante et ses dimensions au Web Worker et récupérer de sa part les coordonnées du rectangle à dessiner. Il peut être judicieux également de manipuler les couleurs d'image dans le Web Worker, comme auparavant, et de les afficher dans une couleur différente afin de vérifier le résultat de la segmentation.

© 2012 Pearson France – HTML5 <video> – Silvia Pfeiffer

Le code des `postFrame()` et `drawFrame()` dans la page web principale est fourni dans le Listing 7.7. Le reste de la page web est identique au Listing 7.2. Le code pour le Web Worker contient une grande partie du Listing 7.6, dont l'initialisation et la fonction `toGray()`, une fonction pour gérer l'événement `onmessage`, recevoir les arguments de message de la page web principale et reposter l'image et les quatre coordonnées à la page principale. Nous laissons le soin au lecteur de réaliser l'implémentation complète, qui peut sinon être téléchargée à l'adresse mentionnée dans la préface de ce livre.

Listing 7.7 : Segmentation des pixels vidéo à l'aide d'un Web Worker

```
function postFrame() {
    w = 320;
    h = 160;
    sctxt.drawImage(video, 0, 0, w, h);
    frame = sctxt.getImageData(0, 0, w, h);
    arg = { frame: frame, height: h, width: w }
    worker.postMessage(arg);
}
function drawFrame (event) {
    msg = event.data;
    outframe = msg.frame;
    if (video.paused || video.ended) { return; }
    context.putImageData(outframe, 0, 0);
    // Dessiner un rectangle sur le canevas
    context.strokeRect(msg.x, msg.y, msg.w, msg.h);
    calls += 1;
    setTimeout(function () { postFrame(); }, 0);
}
```

Examinons les performances de cet algorithme dans les différents navigateurs. Le Tableau 7.3 présente les différences entre les implémentations avec et sans Web Worker. Comme auparavant, les chiffres indiqués correspondent au nombre d'images que le canevas a peintes pour la vidéo d'exemple de quatre secondes. Le programme tente toujours de peindre autant d'images que possible dans le canevas. Sans Web Workers, cela se produit dans le thread principal et l'ordinateur opère aussi intensément qu'il le peut. Avec des Web Workers, le thread principal peut différer la fonction `postMessage()` sans affecter les performances du thread principal. Il peut donc transmettre moins d'images à gérer au Web Worker.

Tableau 7.3 : Performances des navigateurs sans (à gauche) et avec (à droite) Web Workers pour la segmentation de mouvement

Firefox	*Safari*	*Chrome*	*Opera*
36 22 (WW)	35 29 (WW)	62 50 (WW)	76 70 (WW)

© 2012 Pearson France – HTML5 <video> – Silvia Pfeiffer

La différence entre le nombre d'images fournies au Web Worker et celui qui se trouve traité quand il n'y a qu'un seul thread est la plus réduite dans Safari et Opera. Dans Firefox, le Web Worker s'exécute assez lentement, en ne traitant qu'un petit nombre d'images.

Le recours aux Web Workers libère les threads principaux de Firefox et de Chrome et permet à la vidéo de s'exécuter de manière fluide à nouveau. Le seul navigateur qui bataille encore et dont la lecture reste saccadée est Opera. Comme il n'utilise pas de répartition des threads appropriée pour les Web Workers, on pouvait s'attendre à ce résultat.

Notez que la vidéo s'exécute sur le thread principal, alors que le canevas est alimenté depuis le Web Worker et que nous ne mesurons que les performances du Web Worker. Malheureusement, il n'est pas possible de mesurer les performances de l'élément vidéo en indiquant un nombre d'images lues. Une API de statistiques est cependant en cours de préparation pour les éléments multimédias dans le WHATWG, qui fournira cette fonctionnalité une fois implémentée dans les navigateurs.

Détection de visage

Pour déterminer la présence d'un visage dans une image, la méthode consiste généralement à détecter la couleur de peau et à analyser la forme créée par cette couleur[1]. Nous allons ici suivre la première étape de cette technique simple de détection de visage, en identifiant des zones de couleur de peau. Pour cela, nous allons combiner plusieurs des algorithmes présentés précédemment dans ce chapitre.

L'utilisation directe de couleurs RVB n'est pas très utile pour détecter la couleur de peau, parce qu'il existe de nombreuses tonalités possibles. La proportion relative des couleurs RVB peut cependant aider en grande partie à résoudre cette difficulté. La détection de couleur de peau s'appuie normalement sur l'utilisation de couleurs RVB normalisées. Voici l'une des formules possibles :

```
base = R + V + B
r = R / base
v = V / base
b = B / base
```

avec

```
(0.35 < r < 0.5) ET (0.2 < v < 0.5) ET (0.2 < b < 0.35) ET (base > 200)
```

Cette équation identifie la plupart des pixels généralement perçus comme correspondant à la couleur de la peau, mais elle produit aussi des faux positifs. Elle est plus fiable avec les peaux claires qu'avec les peaux foncées, mais en réalité plus

1. Voir **http://fr.wikipedia.org/wiki/D%C3%A9tection_de_visage**.

© 2012 Pearson France – HTML5 <video> – Silvia Pfeiffer

sensible aux différences d'éclairage qu'aux tons de peau. Vous pouvez vous documenter afin de trouver d'autres méthodes plus sophistiquées[1]. Pour les besoins de notre démonstration, nous nous contenterons de cette procédure simple.

On peut filtrer les pixels faux positifs en réalisant une analyse de forme des zones détectées et en identifiant différentes zones comme les yeux et la bouche. Nous n'exécuterons pas ces étapes supplémentaires ici et nous contenterons d'appliquer l'équation précédente ainsi que la segmentation présentée à la section précédente afin de retrouver les zones susceptibles de correspondre à des visages.

Le Listing 7.8 présente le code de la page web principale. Le Listing 7.9 présente le Web Worker associé.

Listing 7.8 : Méthode de détection de visage du thread principal utilisant un Web Worker

```
<video controls height="270px" width="480px" >
    <source src="video5.mp4" type="video/mp4">
    <source src="video5.ogv" type="video/ogg">
    <source src="video5.webm" type="video/webm">
</video>
<canvas width="400" height="300" style="border: 1px solid black;"> </canvas>
<canvas id="scratch" width="400" height="300" style="display: none;"> </canvas>
<script>
    window.onload = function() { initCanvas(); }
    var worker = new Worker("worker.js");
    var context, video, sctxt, canvas;
    function initCanvas() {
            video = document.getElementsByTagName("video")[0];
            canvas = document.getElementsByTagName("canvas")[0];
            context = canvas.getContext("2d");
            scratch = document.getElementById("scratch");
            sctxt = scratch.getContext("2d");
            video.addEventListener("play", postFrame, false);
            worker.addEventListener("message", drawFrame, false);
    }
    function postFrame() {
            w = 320;
            h = 160;
            sctxt.drawImage(video, 0, 0, w, h);
            frame = sctxt.getImageData(0, 0, w, h);
            arg = { frame: frame, height: h, width: w }
            worker.postMessage(arg);
```

1. Comme exemple d'approche améliorée, voir **http://www.icgst.com/GVIP05/papers/P1150535201.pdf**.

© 2012 Pearson France – HTML5 <video> – Silvia Pfeiffer

```
    }
    function drawFrame (event) {
            msg = event.data;
            outframe = msg.frame;
            context.putImageData(outframe, 0, 0);
            // Dessiner un rectangle sur le canevas
            context.strokeRect(msg.x, msg.y, msg.w, msg.h);
            if (video.paused || video.ended) { return; }
            setTimeout(function () { postFrame(); }, 0);
    }
</script>
```

Listing 7.9 : Web Worker pour la méthode de détection de visage du Listing 7.8

```
// Initialisation pour la segmentation
var width = 320;
var height = 160;
var region = new Array (width*height);
var index = 0; region[0] = {};
region[0]['weight'] = 0;
region[0]['x1'] = 0;
region[0]['x2'] = 0;
region[0]['y1'] = 0;
region[0]['y2'] = 0;
function isSkin(r,g,b) {
    base = r + g + b;
    rn = r / base;
    gn = g / base;
    bn = b / base;
    if (rn > 0.35 && rn < 0.5 && gn > 0.2 && gn < 0.5 && bn > 0.2 && bn < 0.35 &&
base > 250) { return true; } else { return false; } }
    onmessage = function (event) {
            // Récupérer les données d'image
            var data = event.data;
            var frame = data.frame;
            var height = data.height;
            var width = data.width;
            // Initialiser les champs de zone et la couleur dans les pixels de
mouvement
            for (x = 0; x < width; x++) {
                    for (y = 0; y < height; y++) {
                            i = x + width*y;
                            if (i != 0) region[i] = {};
                            region[i]['weight'] = 0;
                            // Est-ce une couleur de peau?
```

```
                                if (isSkin(frame.data[4*i],frame.data[4*i+1],frame.
data[4*i+2])) {
                                        // Couleur dans les pixels très différents
                                        frame.data[4*i+0] = 0;
                                        frame.data[4*i+1] = 100;
                                        frame.data[4*i+2] = 255;
                                        // Initialiser les zones
                                        region[i]['weight'] = 1;
                                        region[i]['x1'] = x;
                                        region[i]['x2'] = x;
                                        region[i]['y1'] = y;
                                        region[i]['y2'] = y;
                                }
                        }
                }
                // Segmentation
                for (x = 0; x < width; x++) {
                        for (y = 0; y < height; y++) {
                                i = x + width*y;
                                if (region[i]['weight'] > 0) {
                                        // check the neighbors
                                        for (xn = Math.max(x-2,0); xn <= Math.
min(x+2,width-1); xn++) {
                                                for (yn = Math.max((y-2),0);
yn <= Math.min((y+2),(height-1)); yn++) {
                                                        j = xn + width*yn;
                                                        if (j != i) {
                                                                if (region[j]
['weight'] > 0) {
                                                                region[i]
['weight'] += region[j]['weight'];
                                                                region[i]['x1'] =
Math.min(region[i]['x1'], region[j]['x1']);
                                                                region[i]['y1'] =
Math.min(region[i]['y1'], region[j]['y1']);
                                                                region[i]['x2'] =
Math.max(region[i]['x2'], region[j]['x2']);
                                                                region[i]['y2'] =
Math.max(region[i]['y2'], region[j]['y2']);
                                                        }
                                                }
                                        }
                                }
                        }
                }
        }
```

```
        // Trouver l'un des pixels les plus lourds (l'un des groupes les plus grands)
        max = 0;
        index = 0; // reset
        for (x = 0; x < width; x++) {
                for (y = 0; y < height; y++) {
                        i = x + width*y;
                        if (region[i]['weight'] > max) {
                                max = region[i]['weight'];
                                index = i;
                        }
                }
        }
        // Retransmettre les données d'image et de rectangle au thread principal
        arg = { frame: frame,
        x: region[index]['x1'],
        y: region[index]['y1'],
        w: (region[index]['x2'] - region[index]['x1']),
        h: (region[index]['y2'] - region[index]['y1']) } postMessage(arg);
    }
```

Vous remarquerez que le code est pour l'essentiel identique à celui de la détection de zone de mouvement, à l'exception d'une partie du travail servant à conserver les images des différences (qui peut maintenant être supprimée) et la fonction `toGray()`, qui a été remplacée par une fonction `isSkin()`.

Pour notre exemple, nous avons choisi une vidéo sous licence Creative Commons intitulée *Science Commons*. Une partie des images analysées est présentée à la Figure 7.6. Elles sont toutes affichées en temps réel pendant la lecture de la vidéo.

Figure 7.6
Résultats de la détection de visage sur une vidéo dans Firefox, Safari, Opera et Chrome (de gauche à droite).

© 2012 Pearson France – HTML5 <video> – Silvia Pfeiffer

Ces exemples montrent quand l'algorithme de couleur de peau fonctionne bien : les deux captures d'écran d'Opera et de Chrome révèlent que la segmentation de zone n'a pas repéré les visages, mais les mains, qui occupent des zones plus larges.

La Figure 7.7 présente quelques exemples de faux positifs.

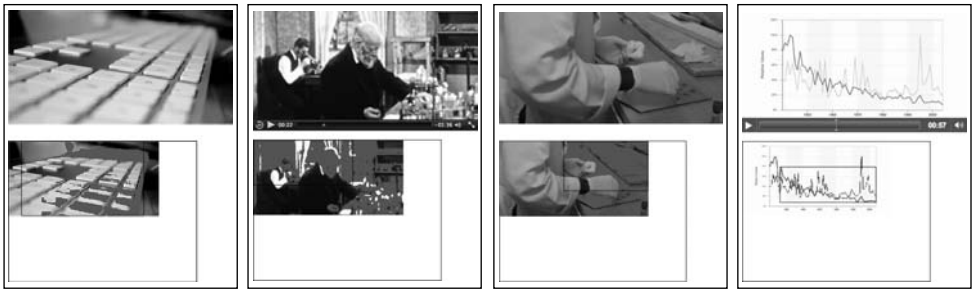

Figure 7.7
Faux positifs de détection de visage utilisant la couleur de peau dans Firefox, Safari, Opera et Chrome (gauche à droite).

L'affichage des résultats d'analyse dans le canevas sous la vidéo se dégrade rapidement quand les tâches de calcul deviennent de plus en plus complexes, comme celles traitées dans ce chapitre. Si vous souhaitez préserver la qualité de lecture vidéo en réalisant l'analyse à l'arrière-plan, il est préférable d'abandonner le coloriage des pixels dans le canevas et de ne peindre que les recouvrements rectangulaires pour les zones détectées sur la vidéo d'origine. Vous obtiendrez de meilleures performances pour le thread du Web Worker.

En résumé

Au cours de ce chapitre, nous avons vu comment utiliser des Web Workers pour prendre en charge une partie du traitement vidéo lorsqu'il est exécuté en temps réel à l'intérieur du navigateur web. Nous avons vu comment y faire appel pour des méthodes de traitement vidéo simples, comme la coloration en tons sépia, et nous avons constaté que pour des tâches élémentaires de ce type, la surcharge liée à la génération d'un thread et aux allers-retours des données passées dans les messages contrebalançait l'allègement obtenu grâce à la délégation du travail.

Nous avons également analysé l'usage des Web Workers pour des tâches plus intenses, comme la détection de mouvement, la segmentation de zones et la détection des visages. Dans ce cas, l'interêt du Web Worker est qu'il permet de déléguer la charge de traitement afin de décharger le thread principal qui, ainsi libéré, peut rester réactif aux actions de l'utilisateur. L'inconvénient est que le navigateur ne fonctionne pas aussi bien que la partie liée au traitement vidéo, si bien que le Web Worker en vient rapidement à manquer d'images à traiter. La réactivité globale

© 2012 Pearson France – HTML5 <video> – Silvia Pfeiffer

accrue du navigateur se fait donc au prix d'une cadence d'images moindre dans le traitement vidéo.

Les Web Workers sont plus particulièrement productifs pour les tâches qui ne requièrent pas d'échanges de données trop fréquents entre le thread principal et celui du Web Worker. Pour que les Web Workers soient plus productifs pour le traitement vidéo, il faudrait qu'ils possèdent un moyen plus simple d'accéder aux images vidéo sans échanger de messages avec le thread principal.

La plupart des algorithmes utilisés dans ce chapitre étaient très simples, mais le but n'est pas de vous enseigner ici des techniques de traitement de l'image. Trouvez un bon livre sur l'analyse vidéo en temps réel, renseignez-vous sur les dernières avancées dans le domaine et amusez-vous. Aujourd'hui, vous pouvez faire tout cela dans votre navigateur web, en temps réel. Les Web Workers pourront vous aider à éviter que ces opérations perturbent la vitesse d'affichage de votre page web principale.

© 2012 Pearson France – HTML5 <video> – Silvia Pfeiffer