Programmation Web Avancée

Cours 4
Élément Canvas
Web Storage

kn@lri.fr

UNIVERSITÉ PARIS SUD

Comprendre le monde, construire l'avenir

université PARIS-SACLAY

Plan

Historique

L'élément **canvas** a été introduit par Apple dans le moteur Webkit (utilisé dans Safari, puis Chrome, ...).

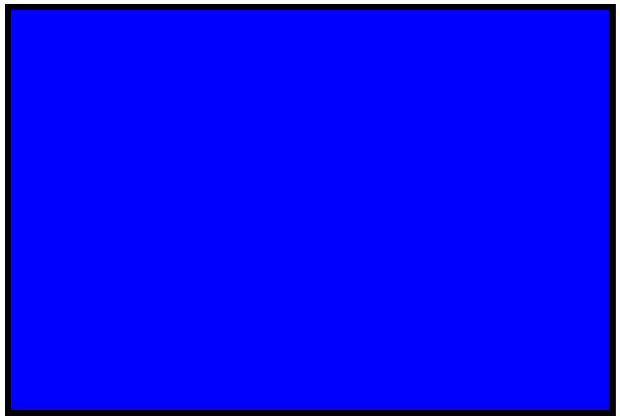
But : disposer d'une zone dans laquelle effectuer du dessin (vectoriel).

Standardisé dans HTML 5 et adopté dans les autres navigateurs

Canvas en HTML

L'élément ***canvas*** définit une zone de dessin. On peut en définir la taille via des attributs **`width`** et **`height`**, et la couleur de fond via l'attribut CSS style :

```
<canvas id="can1" width="300" height="200"
        style="border:2pt solid black; background:blue;">
</canvas>
```



Le *contenu* du canvas est ensuite manipulable via Javascript.

Contexte

En Javascript, les objets Canvas possèdent deux contextes :

- Un contexte ***2d*** pour faire du dessin vectoriel
- Un contexte ***webgl*** pour faire de la 3D

On ne s'intéresse dans cette introduction qu'au contexte **2d**.

```
let canvas = document.getElementById("can1");
let context = canvas.getContext("2d");
```

L'objet **context** expose des méthodes permettant de dessiner selon le modèle choisi (2D ou 3D).

Dans le contexte **2d** le point (0,0) correspond au coin supérieur gauche du canvas.

Chemin

Les canvas 2D supportent des primitives de dessin vectoriel. On dessine des *chemins* que l'on peut ensuite tracer ou remplir. Sur l'objet **`context`** :

`.beginPath()` : commence un chemin

`.moveTo(x,y)` : bouge le curseur au point (x,y)

`.lineTo(x,y)` : trace une ligne entre le curseur et le point (x,y) et positionne le curseur au point (x,y)

`.closePath()` : (optionnel) ferme le chemin courant en revenant au point de départ

`.rect(x,y,w,h)` : dessine un rectangle au point (x,y), d'une largeur de w et d'une hauteur de h

`.arc(x,y,rad, startAngle, stopAngle, ccw)` : dessine un arc de cercle au point (x,y), d'un rayon de **`rad`** entre les angles **`startAngle`** et **`stopAngle`** (en radians). Si **`ccw`** vaut vrai, l'arc est tracé dans le sens anti-horaire, sinon dans le sens horaire

`.stroke()` : dessine le chemin courant

`.fill()` : remplis le chemin courant

Style des chemins

On peut définir le style du trait ainsi que le style du remplissage. Sur l'objet **`context`** :

`.strokeStyle = c` : permet de définir la couleur de trait à c
`.fillStyle = c` : permet de définir la couleur de remplissage à c
`.lineWidth = n` : permet de définir l'épaisseur du trait (en pixels)

Exemple

```
context.beginPath();
context.moveTo(10,10);
context.lineTo(110,10);
context.lineTo(60,90);
context.closePath();
context.stroke();
context.beginPath();
context.rect(200,50, 200, 10);
context.fillStyle = '#ffff00';
context.lineWidth = 5;
context.fill();
context.stroke();
context.beginPath();
context.strokeStyle = '#ff0000';
context.arc(600, 30, 25, 0, Math.PI * 1.5, false);
context.stroke();
```

Texte

On peut aussi tracer sur un canvas

`.strokeText(txt, x, y, maxWidth)` : trace le texte **`txt`** aux coordonnées (x,y), en prenant au plus **`maxWidth`** pixels de large.

`.fillText(txt, x, y, maxWidth)` : remplit le texte **`txt`** aux coordonnées (x,y), en prenant au plus **`maxWidth`** pixels de large.

`.font = desc` : change la police courante avec la chaîne de description **`desc`**

`.measureText(txt)` : renvoie un objet décrivant la taille du texte donné

```
context.font = "80px Arial";
context.strokeText("Empty", 10, 80);
var dim = context.measureText("Empty");
context.fillText("Full", 40 + dim.width, 80);
```

Transformations

L'intérêt d'avoir une représentation vectorielle des images est de pouvoir lui appliquer des transformations sans perte. Sur l'objet **`context`** :

`.scale(hScale, vScale)` : Zoom le canvas d'un certain facteur (horizontal et vertical).
`.rotate(angle)` : Applique une rotation de **`angle`** radians au canvas.
`.translate(x, y)` : Fait du point (x,y) la nouvelle origine du canvas (i.e., l'ancien coin supérieur gauche est translaté du vecteur (x,y)).

Plan

Principe

Pour animer un canvas, il suffit (et il faut) redessiner (tout) le contenu à intervales réguliers !

```
let context = document.getElementById("can4").getContext("2d");
let pos = 0;
let dir = 25;
function drawBox() {

 if (redraw)  context.clearRect(0,0, 500, 100);

 pos += dir;
 if (pos < 0 || pos > 400) dir = -dir;
 context.beginPath();
 context.fillStyle = '#ff0000';
 context.fillRect(pos,0, 100, 100); //alias pour rect/fill
};
setInterval(drawBox, 16);
```

Problèmes

Pour obtenir une animation fluide, il faut essayer de viser 60 images par secondes, soit une image toutes les 16.6 ms!

Tous les calculs à faire entre deux affichage (nouvelles position des objets, nouvelles formes, ...) doivent être fait en moins de 16 ms

`setInterval` n'est pas adapté :

- Pas de garantie que le code est exécuté à intervalle régulier
- **`setInterval`** n'est pas spécifique aux animations. Il continue de s'exécuter, même quand le canvas n'est pas visible ou qu'il va être redessiné.

requestAnimationFrame

Le standard fournit une primitive, `requestAnimationFrame(f)` qui appelle la fonction `f` en lui passant en argument le temps écoulé depuis le début du programme, en millisecondes avec une précision de l'ordre de la nanoseconde.

- La fonction est appelée le plus possible, sans dépasser 60 fois / secondes
- L'heure permet de calculer le delais depuis le dernier appel, et éventuellement de « sauter » une frame pour faire des calculs
- La fonction est appellée au « bon moment » vis à vis du navigateur (i.e. pas appelée quand on redimentionne la fenêtre ou que le canvas n'est pas visible)

Plan

Un peu de physique

On souhaite modéliser des corps célestes (planètes) ainsi que la loi de Newton (attraction universelle). On a besoin des choses suivantes :

- Une petite bibliothèque de vecteurs 2D (avec primitives de base : produit scalaire, somme, ...)
- $\vec{F}_{AB} = \frac{G \cdot m_A \cdot m_B}{|AB|^2} \cdot \vec{u}_{AB}$ (avec G constante gravitationnelle, m_A masse de A, m_B masse de B, $\vec{u}_{AB}$ vecteur unitaire de A vers B
- $\Sigma \vec{F} = m \cdot a$ avec a l'accéleration (dérivée de la vitesse, dérivée seconde de la position)

Principe de la simulation

- On place un certain nombre de planètes dans l'univers
- Toutes les 16ms :
  1. On calcule pour chaque planète la force qu'exerce chaque autre planète sur elle
  2. On additionne toutes les forces ainsi calculées
  3. On en déduit l'accéleration pour chaque planète
  4. L'acélération × 16ms donne la variation de vitesse
  5. Avec la nouvelle vitesse, on calcule la nouvelle position au bout de 16 ms
  6. On efface la planète à l'ancienne position et on la dessine à la nouvelle

Plan

Persistance

Le développement d'application Web moderne (et en particulier mobiles) fait apparaître un problème de *persistance* des données

En effet, on souhaîte pouvoir stocker *l'état du programme* Javascript (pour pouvoir le restaurer plus tard). On peut aussi vouloir stocker des données volumineuses pour pouvoir y accéder « hors connexions ».

Les solutions « naïves » ne fonctionnent pas pour Javascript :

- Impossible en Javascript d'écrire dans des fichiers du répertoire utilisateur (modèle de sécurité de Javascript)
- Utilisation des *cookies* : ne permet pas de stocker des donnéer de grande taille.
- Envoi de données au serveur : nuit à l'interactivité

Storage

L'object ***Storage*** implémente une table associative clé → valeur :

`.setItem(key, value)` : Associe dans l'espace de stockage la valeur **`value`** à la clé **`key`** (toutes deux sont des chaînes de caractères arbitraires).
`.getItem(key)` : Renvoie la valeur associée à la clé **`key`**
`.removeItem(key)` : Supprime la valeur associée à la clé **`key`**
`.length` : Renvoie le nombre de clé associées
`.key(i)` : Renvoie la valeur de la $i^{ème}$ clé

Deux objets de type ***Storage*** sont accessibles :

- ***localStorage*** : stockage à durée de vie illimité
- ***sessionStorage*** : stockage jusqu'à la fin de la session (fermeture de l'onglet ou du navigateur)

Démo

On va implémeter un simple compteur de click. On stocke que le nombre de click à la fois dans le **`localStorage`** et dans le **`sessionStorage`**.

Plan

Accès au données depuis Javascript

Le modèle de sécurité des navigateurs Web impose une politique de même origine pour l'accès à des données (*same origin policy*).

Les scripts de deux pages ne peuvent partager des données que *si ces pages ont la même origine*

Deux pages ont la même origine si elles ont les même *protocole*, *hôte* et *port*.
Si on considère : **`http://www.example.com/dir/page.html`** :

- ✓ ***`http://www.example.com/dir/page2.html`***
- ✓ ***`http://www.example.com/dir2/other.html`***
- ✓ ***`http://user:password@www.example.com/dir2/other.html`***
- ✗ **`http://www.example2.com/dir/page2.html`**
- ✗ **`http://example.com/dir/page2.html`**
- ✗ **`http://www.example.com:8080/dir/page2.html`**
- ✗ **`https://www.example.com/dir/page2.html`**

? **`http://www.example.com:80/dir/page2.html`** (dépend des navigateurs)

Politique de sécurité

Si deux pages ont la même origine, alors elle peuvent se manipuler librement l'une/l'autre (changer leur DOM, enregistrer/supprimer des gestionnaires d'évènements, lire l'espace de stockage et les cookies, ...)

Cela pose de gros problèmes de sécurité et de confidentialité si les deux pages n'ont pas la même origine ⇒ interdit.

Attention : un script a pour origine celle de la page qui le charge et non pas l'URL à laquelle il est stocké