TP 1 : Javascript et HTML5 !

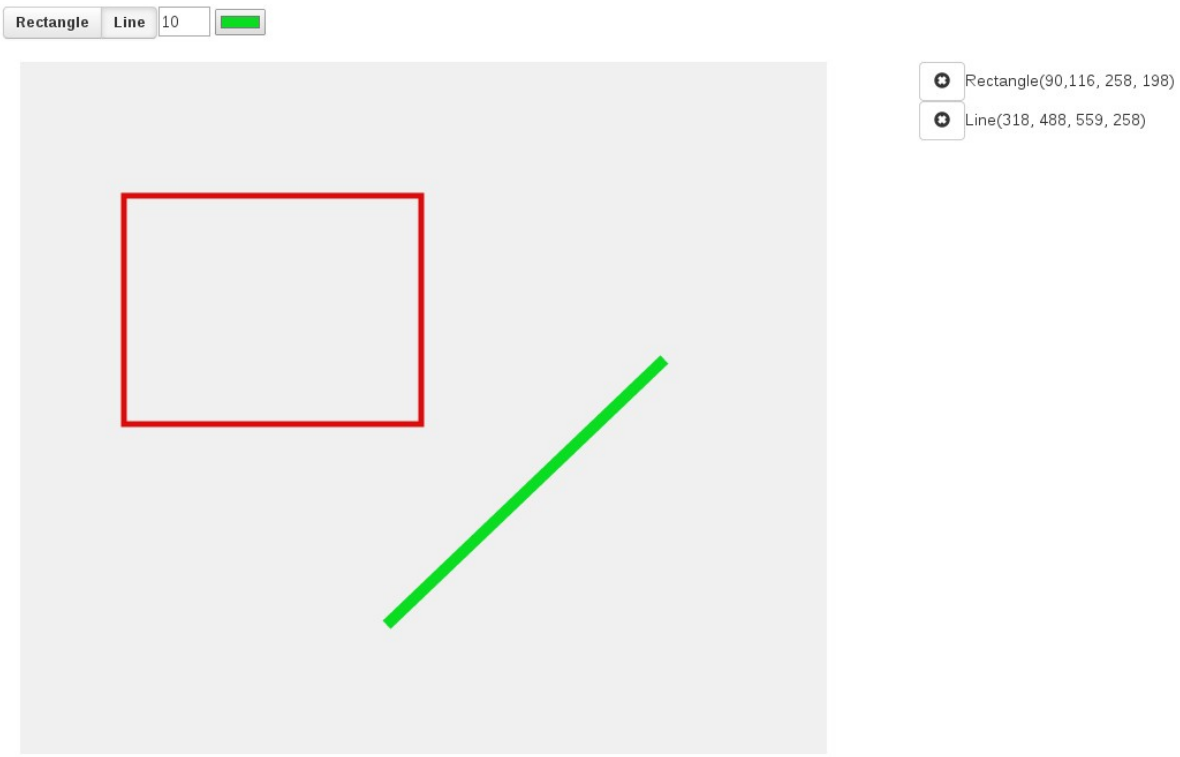
HTML5 (*HyperText Markup Language*) est la dernière version majeure du langage HTML dont la finalisation est en cours mais le support largement réalisé par les navigateurs sérieux actuels (Chrome, Firefox, *etc.*). Il est difficile de parler d'HTML5 sans parler de Javascript, un langage de programmation principalement utilisé dans les pages Web.

Si Javascript est un langage relativement ancien (créé en 1995 chez Netscape), son engouement est relativement récent et est dû principalement à l'avènement des mobiles, du Web en général et des applications Web de plus en plus gourmandes en termes de performances et de fonctionnalités.

L'objectif de ce TP est de s'initier à Javascript au travers du développement d'une application Web HTML5.

1. La réalisation

L'objectif de ce TP consiste à créer une application Web pour faire du dessin vectoriel, *i.e.* pour dessiner des rectangles, des lignes, ainsi que définir leur couleur et leur épaisseur de trait. La figure ci-dessous montre l'état de l'application et donc l'objectif à atteindre.

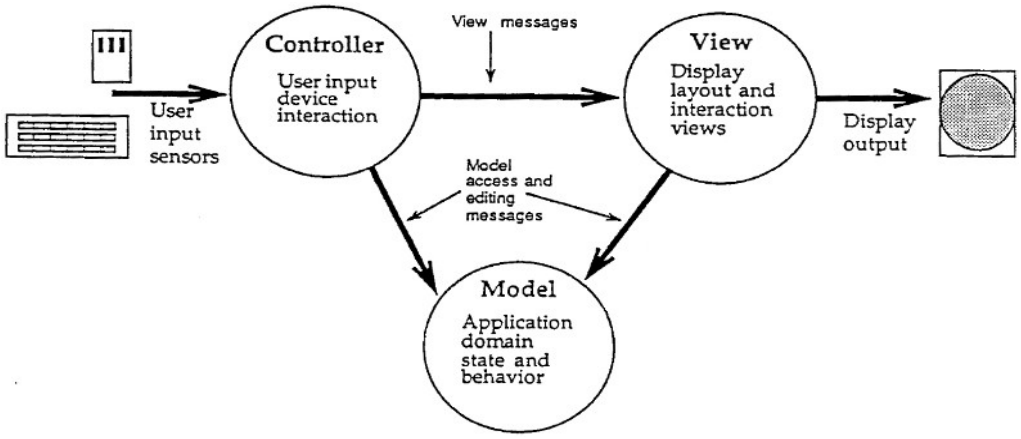


Pour cela, vous allez utiliser le patron d'architecture MVC (Modèle-Vue-Contrôleur) ainsi que l'élément HTML5 *Canvas*.

Le fichier fourni *canvas.css* décrit des styles utilisés par le programme. Ne vous en soucier pas. Le fichier *canvas.html* est un document HTML décrivant la structure du programme : un *Canvas*, des boutons à bascule (*toggle button*), un sélectionneur de couleur (widget HTML5) et un bouton fléché (*spinner*) permettant de choisir l'épaisseur de trait. Ce document importe le document css (ligne 7) et les documents Javascript nécessaires à l'exécution du programme Web (ligne 26-30). Ces documents Javascript vous sont fournis mais sont vides et devront donc être complétés.

Avant de commencer le TP, les deux sections suivantes introduisent les notions nécessaires à sa réalisation.

2. MVC



MVC (*Modèle-Vue-Contrôleur*) est un patron d'architecture IHM. Il sera étudié plus en détails en 4 INFO. Ce patron suggère d'organiser la structure d'un logiciel en 3 composants :

- Le *Modèle* définit le modèle de données du logiciel. Dans notre cas il s'agira de définir les concepts d'un dessin (dessin, forme, *etc.*) et leurs propriétés (couleur, *etc.*). Le modèle ne doit pas dépendre d'une quelconque librairie graphique, il est indépendant de ces possibles représentations graphiques.
- La *Vue* est une représentation graphique possible du modèle. On peut évidemment avoir plusieurs vues d'un même modèle. Dans notre cas, la vue va consister à peindre dans un *canvas* HTML5 les formes du modèle.
- Le *Contrôleur* est la partie interactive du patron. Il a pour but transformer les interactions réalisées par l'utilisateur en commandes allant modifier le modèle.

3. Javascript

Javascript est actuellement *le* langage de programmation de scripts utilisé pour apporter du comportement à des pages Web. Javascript est en fait une implémentation du standard ECMAScript. Un script Javascript s'intègre nativement dans une page HTML de différentes manières possibles dont :
<script src="model.js" type="text/javascript"></script> est une balise HTML permettant d'inclure un script Javascript dans une page.

<script type="text/javascript">alert(prompt("Hello World!"));</script> permet ici d'écrire directement du Javascript dans la balise *script*.

Une instruction Javascript prend la forme suivante :

var *canvas = document.getElementById('myCanvas');*

où *canvas* est déclaré comme étant une variable. Javascript est un langage à typage faible et dynamique : *faible*, car on ne déclare pas le type d'une variable, d'un paramètre ; *dynamique*, car on peut changer le type d'une variable. Par exemple on peut écrire :

var *maVariable = 10 ;*
maVariable = "Fooo!" ;

Vous noterez qu'il est possible d'accéder au contenu d'un page Web *via* la variable globale *document*. On peut également rechercher un élément dans une page Web en utilisant sont *id* défini dans la page HTML :

```
<canvas id="myCanvas"></canvas>
```

Javascript permet de définir des fonctions, par exemple :

```
function getMousePosition(canvas, evt) {
  var rect = canvas.getBoundingClientRect();
  return {
    x: evt.clientX - rect.left,
    y: evt.clientY - rect.top
  };
};
```

Les paramètres ne sont également pas typés. On peut également appeler une fonction sans fournir tous les paramètres. Par exemple :

getMousePosition(monCanvas, monEvt) ; est l'appel classique, tandis que

getMousePosition(monCanvas) ; fonctionne également à la différence que le paramètre *evt* n'aura pas de valeur dans la fonction.

Vous noterez que l'on peut retourner plusieurs valeurs en même temps :

```
var res = getMousePosition(monCanvas, monEvt) ;
```

La variable *res* contiendra 2 valeurs : *x* et *y* comme défini dans la fonction *getMousePosition*. On peut donc afficher ces deux valeurs de la manière suivante :

```
console.log(res.x) ;
console.log(res.y) ;
```

Un tableau s'écrit de la manière suivante :

```
var monTableau = new Array();
```

L'ajout d'un élément dans un tableau :

```
monTableau.push(elt) ;
```

Le parcours d'un tableau :

```
monTableau.forEach(function(eltDuTableau) { console.log(eltDuTableau) });
```

Étant donnée l'énumération suivante :

```
var editingMode = { rect: 0, line: 1 };
```

L'instruction *switch* s'utilise de la manière suivante.

```
switch(this.currEditingMode){
  case editingMode.rect: {
    break;
  }
  case editingMode.line: {
    break;
  }
}
```

Javascript est un langage de programmation orienté objet. Cependant il ne permet pas de définir des classes mais des prototypes. Un prototype est un objet à partir duquel il est possible d'en créer de nouveaux (des duplicata). Pour ce TP, il n'est pas nécessaire de comprendre finement cette nuance. D'ailleurs, dans une optique de simplicité nous parlerons de classes par la suite. Donc, la définition d'une classe (en fait d'un objet mais nous allons utiliser le terme *classe* pour simplifier les explications) se fait de la manière suivante :

```
function Humain(poids, taille) {
  this.poids = poids;
  this.taille = taille;
  this.indiceMasseCorporelle = function() {
    return this.poids / this.taille*this.taille;
  }.bind(this) ;
};
```

Il s'agît donc d'une fonction dont le nom commence par une majuscule. Cette fonction est en fait le constructeur de l'objet pouvant prendre des paramètres (ici, *poids* et *taille*). Il est recommandé de déclarer les attributs d'une classe (*this.poids* et *this.taille*). La variable *this* correspond au contexte de l'objet courant.
Une classe peut avoir des fonctions (ici, *indiceMasseCorporelle*). Une fonction d'une classe peut manipuler les attributs de celle-ci. Le code *bind(this)* permet de lier la fonction à la classe : le *this* utilisé dans la fonction devient le *this* de l'objet. Ne l'oubliez pas !
L'instanciation d'un objet se fait de la manière suivante :
var monHumain = **new** Human(60, 170) ;
Il est possible de définir des sous-classes :

```
function Femme(poids, taille) {
  Humain.call(this, poids, taille);
};
Femme.prototype = new Human();
```

La ligne *Femme.prototype = new Human();* permet de définir *Femme* comme étant un sous-type de *Humain*.
La ligne *Humain.call(this, poids, taille);* permet d'appeler le constructeur de la super-classe (*Humain*). On peut voir cela comme le *super* de Java.
Il est possible de rajouter des fonctions après la définition d'une classe. Vous allez utiliser ce principe pour rajouter dans le modèle une fonction d'affichage. Par exemple :

```
Humain.prototype.indiceMasseGrasse = function() {
// Pour appeler l'éventuelle opération de la classe supérieure (le « super ») :
// NomDeLaClasseSupérieure.prototype.indiceMasseGrasse.call(this) ;
 return this.indiceMasseCorporelle()*1.2 + this.age*0.23 – 5.4 ;
};
```

Cela rajoute dans la classe *Humain* la fonction *indiceMasseGrasse*. Cet ajout peut être effectué n'importe où après la déclaration de la classe.

Le *canvas* est un objet HTML5 sur lequel il est possible de brancher des fonctions pour notamment récupérer les événements souris :

```
canvas.addEventListener('mousedown', this.maFctGérantLaPression, false);
canvas.addEventListener('mousemove', this.maFctGérantLeDéplacement, false);
canvas.addEventListener('mouseup', this.maFctGérantLeRelâchement, false);
```

Vous allez devoir utiliser ces 3 lignes pour brancher l'interaction glisser-déposer (*cf.* section suivante) au *canvas* de la page Web.
Nous avons uniquement présenté les concepts Javascript nécessaires pour ce TP. Voici quels liens indispensables pour une maîtrise plus approfondie :
https://developer.mozilla.org/en-US/docs/Web/JavaScript

https://github.com/airbnb/javascript

https://en.wikibooks.org/wiki/JavaScript
http://www.html5canvastutorials.com/
De plus, la plupart des navigateurs actuels fournissent des outils de développement très utiles ! Par exemple sous Firefox, le menu « Développeur Web » dans le menu « Outils » référence un ensemble d'outils intégrés dans Firefox, notamment la « Console web » à ouvrir pour le TP.

4. Interaction : le glisser-déposer (Drag-n-Drop)

Chose peu commune, vous allez commencer l'éditeur par l'implémentation d'une interaction homme-machine courante, le *Drag-n-Drop* (glisser-déposer, DnD) dans le fichier interaction.js. Le DnD consiste en la suite d'événements suivante s'effectuant à l'aide de la souris : une pression (santé !), des déplacements, un relâchement. Le DnD sera utilisé dans notre application Web pour dessiner des rectangles et des lignes.

Q. 1 Créer une classe DnD contenant les 4 attributs suivants initialisé à 0 : les coordonnées de la position initial du DnD ; celles de la position finale. Ces attributs seront utilisés plus tard lors de la création de formes.

Q. 2 Déclarer 3 fonctions à cette classes correspondant aux 3 types d'événements à gérer. Pensez à lier (*bind*) chaque fonction à la classe. Ces fonctions prennent en entrée un attribut *evt* correspondant à l'événement produit lors de l'utilisation de la souris. Un tel événement possède les attributs *x* et *y*.

Q. 3 Implémenter ces fonctions. Elles ont pour but de définir la valeur des attributs. Cela va nécessiter de définir un nouvel attribut dans la classe car le code des fonctions liées aux déplacements et au relâchement doit être exécuté uniquement si une pression a été effectuée au préalable. Utiliser la fonction *getMousePosition* qui va placer le point de l'événement relativement à la position du *canvas*.

Q. 4 Enregistrer chaque fonction auprès du *canvas* (*addEventListener*).

Q. 5 Appeler dans chacune des 3 fonctions *console.log* pour afficher dans la console Javascript de votre navigateur les coordonnées de chaque événement lors de l'exécution de l'interaction. Vous pouvez maintenant tester le bon fonctionnement de l'interaction en ouvrant la page canvas.html dans votre navigateur, en affichant la console Web et en exécutant un DnD sur la zone de dessin (en gris).

5. Le modèle

Vous allez maintenant développer le modèle (au sens MVC) de l'application Web. Ce modèle doit représenter un *dessin* se composant de *formes* (utilisez pour cela *Array*), une forme étant soit un *rectangle*, soit une *ligne*. Une forme possède une *couleur* et une *épaisseur* de trait. Un rectangle possède des *coordonnées de son point haut-gauche*, une *largeur* et une *hauteur*. Une ligne possède les *coordonnées de ses deux points*.

Q. 6 Implémenter les 4 classes nécessaires pour définir le modèle dans le fichier *model.js*. Pensez à dé-commenter les lignes nécessaires dans le fichier *main.html*.

6. La vue

La vue est *une* représentation graphique possible du modèle. Dans notre cas, le modèle est déjà hautement graphique et nous allons donc feinter pour minimiser le développement à faire grâce à la « flexibilité » de Javascript. En effet, Javascript permet de rajouter dans une classe déjà définie des fonctions.

Q. 7 Utiliser cette fonctionnalité pour ajouter les fonctions d'affichage (fonction *paint*) dans chacune des classes. La fonction *paint* de la classe *Forme* configurera juste la couleur et l'épaisseur du trait du contexte du *canvas*. Ces fonctions *paint* prendront donc en paramètre le contexte du *canvas*. La fonction *paint* de la classe *Dessin* peindra la fond du *canvas* en gris clair (copier-coller les lignes 10 et

11 du fichier main.js).

Q. 8 Tester votre code avec le code mis en commentaire dans le fichier main.js.

7. Le contrôleur

Vous allez maintenant rendre l'application interactive en définissant un interacteur. Un interacteur est une sorte d'outil que l'utilisateur manipule pour réaliser des actions. Dans notre cas, l'interacteur sera un crayon (*Pencil* dans le fichier controler.js) que l'utilisateur va manipuler pour ajouter des formes au dessin. Comme vous pouvez le constater le crayon est paramétrable (couleur, épaisseur, type de forme).

Q. 9 La première étape consiste à lier une interaction DnD à Pencil. Dans chacune des 3 fonctions de DnD, ajouter un appel aux fonctions *interactor.onInteractionStart(this);*, *interactor.onInteractionUpdate(this);*, *interactor.onInteractionEnd(this);*. Ces fonctions n'existent pas encore mais elles ont pour but de notifier l'interacteur d'une modification de l'interaction.

Q. 10 Implémenter ces 3 fonctions dans la classe *Pencil*. Elles devront créer une forme (en utilisant l'attribut *this.currentShape*, la forme créée sera un *Rectangle* ou *Ligne* en fonction de l'attribut *this.currEditingMode*), la mettre à jour lorsque l'utilisateur bouge la souris et l'ajouter au dessin lors du relâchement. N'oubliez pas d'appeler la fonction *paint* de la classe *Dessin* pour mettre à jour la vue à chaque étape.

Q. 11 Pour modifier le style des formes créées, l'utilisateur doit pouvoir utiliser les widgets fournis (boutons et *spinner*). Dans la classe *Pencil*, lier chacun de ces widgets à une fonction modifiant les attributs de la classe *Pencil* (*this.currLineWidth*, *this.currColour* et *this.currEditingMode*).

8. Liste des modifications

Vous allez compléter l'application avec une liste des modifications apportées au dessin. Cela va permettre de supprimer une forme précédemment dessinée. Pour implémenter cette nouvelle fonctionnalité, il va falloir modifier la page actuelle via Javascript. La liste sera contenue dans l'élément dont l'ID est *shapeList.*

Q. 12 Ajouter une fonction *updateShapeList* dans la vue pour afficher la liste des formes. Chaque forme aura un affichage différent en fonction de son type (ligne, rectangle). Adapter le contrôleur pour appeler cette nouvelle fonction à chaque fois qu'une nouvelle forme est dessinée.

Q. 13 Modifier *updateShapeList* pour ajouter le bouton suivant devant chaque élément de la liste.

```
<button type="button" class="btn btn-default">
    <span class="glyphicon glyphicon-remove-sign"></span>
</button>
```

Q. 14 Définir le comportement du bouton pour supprimer la forme correspondante dans le dessin.

Q. 15 Si vous avez le temps, compléter l'éditeur avec la création d'autres formes (ellipse, polygone, *etc.*), d'autres paramètre (style de la ligne, *etc.*) ou bien des boutons undo/redo.