R5.A.06 - Prog multimédia - CM n°1

Pierre Nerzic

septembre 2023

Introduction

Pierre Nerzic

Programme national

Le cours est intitulé *Sensibilisation à la programmation multimédia : Manipulation d'images 2D, 3D ; colorimétrie*

C'est à dire ?

Le PPN est très vague et ne cible pas des compétences typiques du diplôme, en particulier la programmation.

Le volume horaire, les prérequis mathématiques et l'utilité professionnelle ne permettent pas du tout d'envisager des cours de traitement et de synthèse d'images. Ce sont deux gros modules de 40h à l'ENSSAT, IAI 2e année. Ici on ne dispose que de 10h.

J'ai donc choisi de vous faire travailler sur la programmation de graphiques dynamiques : dessinés en JS, animés et interactifs, dans un cadre qui s'appelle *data visualization*.

Pierre Nerzic

Visualisation de données

La data visualization ou *dataviz* consiste à traduire des données en graphiques : courbes, histogrammes, schémas, cartes ou autres, de manière à les rendre facilement compréhensibles.

Il existe plusieurs API pour cela, par exemple Chart.js, canvasJS, amCharts, mais on est prisonnier de leurs fonctionnalités.

Ici on s'intéresse à des graphiques non conventionnels, spécifiques aux besoins professionnels. Les données sont susceptibles de changer au cours du temps. On doit donc créer des éléments (lignes, rectangles, courbes...) dynamiquement, c'est à dire après le chargement de la page.

On va utiliser des images vectorielles, SVG, pleinement intégrées dans le DOM d'une page HTML5.

Pierre Nerzic

Concepts du cours

Les images vectorielles SVG sont des balises XML spécifiques dans un document HTML5.

On programme en JavaScript pour les générer et/ou les modifier dynamiquement, simplement en modifiant le DOM.

Des bibliothèques de fonctions permettent de les dessiner et les animer encore plus facilement.

Ainsi, on reprend et on consolide des connaissances que vous avez déjà, tout en faisant quelque chose d'agréable et potentiellement utile.

Pierre Nerzic

Plan simplifié de cet enseignement

1. Découverte des images SVG d'un point de vue programmation
2. Modification du DOM en JavaScript
3. Utilisation de GSAP pour animer ces images
4. Utilisation de d3.js pour dessiner des diagrammes complexes

Ce cours = points 1 à 3, le cours n°2 = point 4.

Les 4 TD et 2 TP suivent ce plan. Les TD = découverte guidée des concepts, les deux TP = mini-projets notés à rendre sur Moodle.

NB : c'est la première année de ce cours, et il y aura sûrement des ajustements.

Ce cours est le début de R5.C.04 pour les étudiants du parcours C.

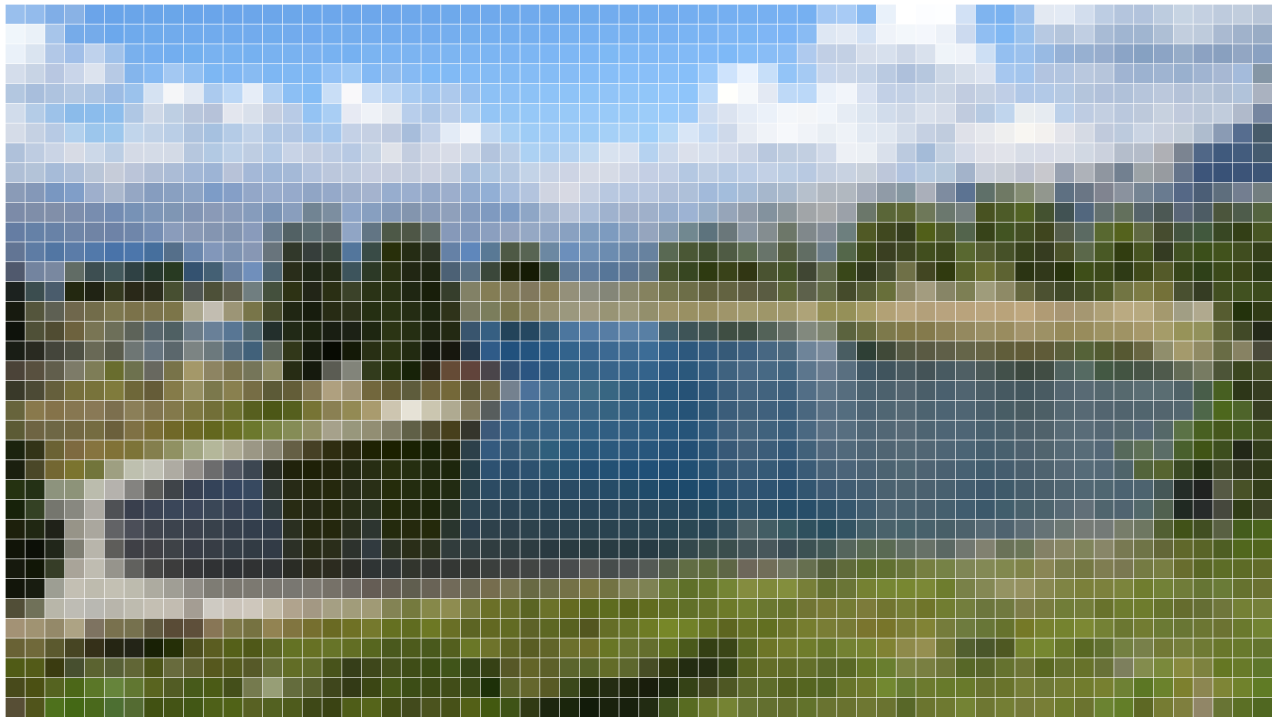
Images vectorielles SVG

Pierre Nerzic

Deux sortes d'images

Les ordinateurs gèrent deux sortes d'images :

- Images matricielles (*raster* ou *pixmap*) : elles sont définies par un tableau rectangulaire de pixels



Pierre Nerzic

Deux sortes d'images, suite

- Images vectorielles : elles sont définies par des figures géométriques : points, lignes, rectangles, ellipses, etc. portant des propriétés ex: position, dimensions, épaisseur, couleur, transparence. . .

Pierre Nerzic

Utilité des images vectorielles

Applications :

- pas du tout adapté à la photographie !
- graphiques géométriques : plans, schémas, logos, polices, icônes (ex: *Font Awesome*)
- modélisations de données : cartographie, CAO...
- structure interne de documents : postscript, pdf...

parce que :

- qualité indépendante de la taille d'affichage (aucun crénelage)
- coordonnées des éléments = nombres réels, indépendants de toute notion de pixel
- taille de fichier réduite comparée à une image raster équivalente
- relativement simples à produire par logiciel

Redimensionnement d'images

Image vectorielle

Image matricielle

Pierre Nerzic

Structure générale d'images SVG

Le format SVG est ouvert et assez simple à comprendre. Il est correctement affiché par de nombreux logiciels.

Une image SVG est définie par un document XML qui peut soit être dans un fichier à part, soit inclus dans un HTML.

```
<svg version="1.0" xmlns="http://www.w3.org/2000/svg">
    ... balises définissant l'image ...
</svg>
```

Le point important est que toutes les balises sous `<svg>` doivent être rattachées au *namespace SVG*. L'attribut `xmlns` fait cela implicitement pour toutes les balises déjà écrites, mais pas pour celles rajoutées par programme. On verra plus loin comment s'en assurer en JavaScript.

Taille de l'image

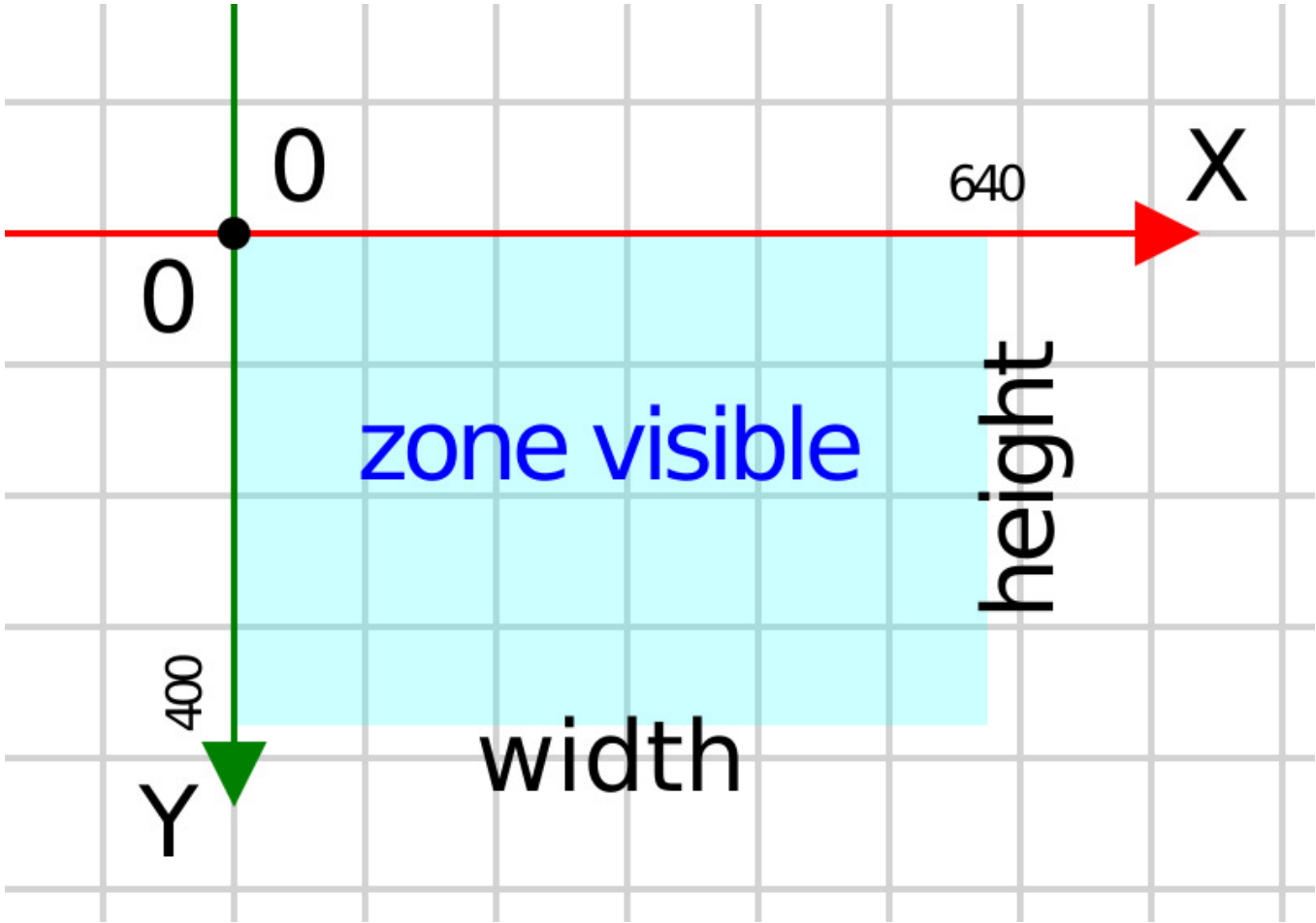
Pour l'affichage sur un écran, on ajoute deux attributs `width` et `height` à la balise `<svg>`. Ils définissent la *taille d'affichage* de l'image en nombre de pixels.

```
<svg version="1.0" xmlns="http://www.w3.org/2000/svg"
    width="640" height="400">
...
```

Dans un document HTML, si ces attributs sont absents, alors la taille est entièrement déterminée par le conteneur de l'image.

Si ces attributs sont présents, ils définissent les limites des coordonnées pour dessiner dans la zone visible. Ce sont des *réels* qui vont de 0 à *width* pour x, et de 0 à *height* pour y.

Coordonnées



Pierre Nerzic

Définition de la zone visible

On peut rajouter l'attribut `viewBox="X Y W H"` pour définir une transformation du système de coordonnées : un décalage d'origine et un changement d'échelle.

```
<svg ... width="640" height="400" viewBox="-10 -5 160 100">
```

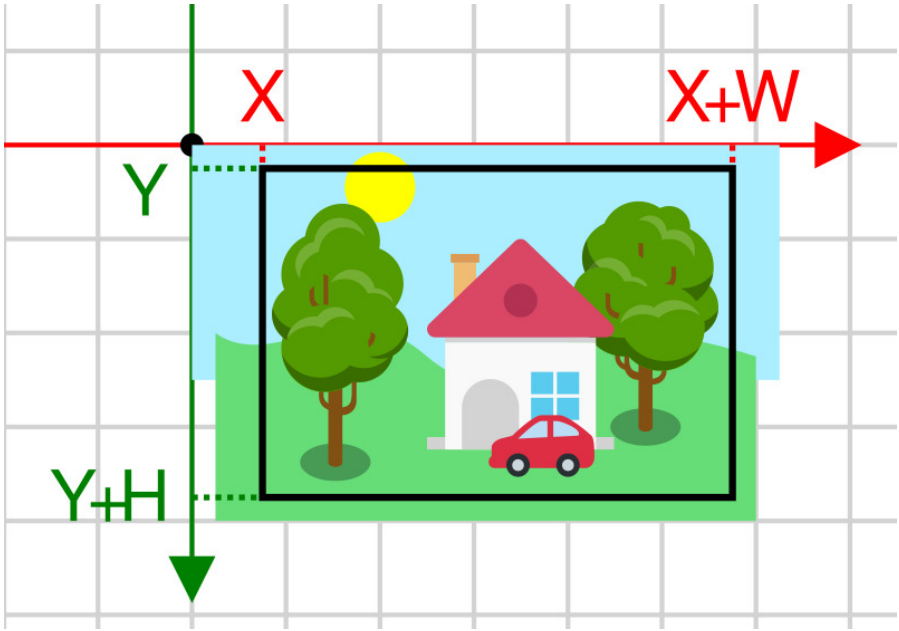
Les valeurs X et Y définissent l'origine des coordonnées de la zone visible, ici en $(-10, -5)$. Les deux nombres suivants, W et H définissent l'étendue de la zone visible à partir de (X, Y).

Dans cet exemple, un point sera visible si son abscisse x est comprise entre -10 et 150, et son ordonnée y entre -5 et 95.

Donc W et H sont liés à *width* et *height*. Généralement, on fait en sorte que $\exists k$ tq $height = k * W$ et $width = k * H$, pour maintenir le rapport largeur/hauteur. Ici $k = 4$

Exemple de *viewBox* (zone visible)

L'attribut `viewBox="X Y W H"` définit un système de coordonnées dans lequel (x, y) est visible si $x \in [X, X + W]$ et $y \in [Y, Y + H]$



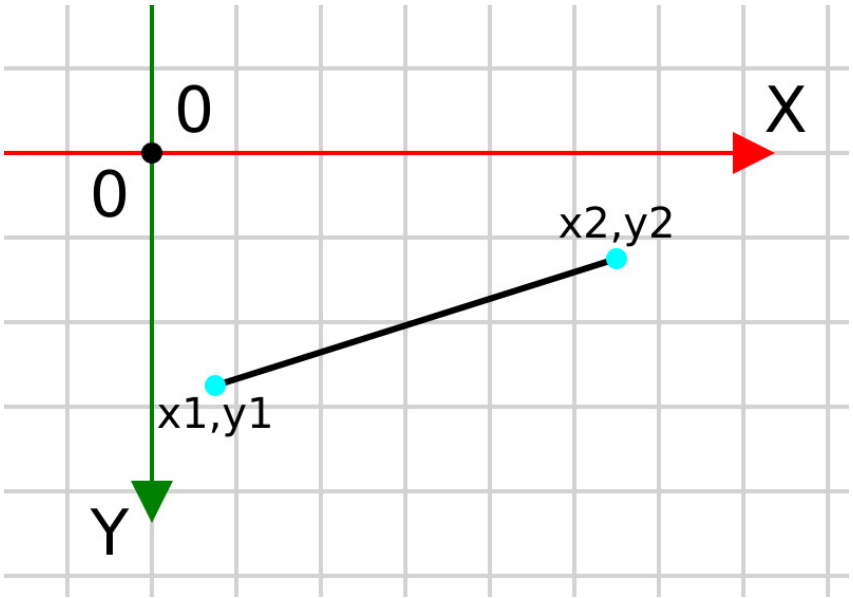
`<svg viewBox="X Y W H">`
affiche seulement ceci :

Pierre Nerzic

Primitives simples

Elles sont définies par les coordonnées de points et/ou des *largeur* et *hauteur* placées dans des attributs. Voir la doc en ligne.

- `<line x1="x1" y1="y1"  x2="x2" y2="y2"/>`

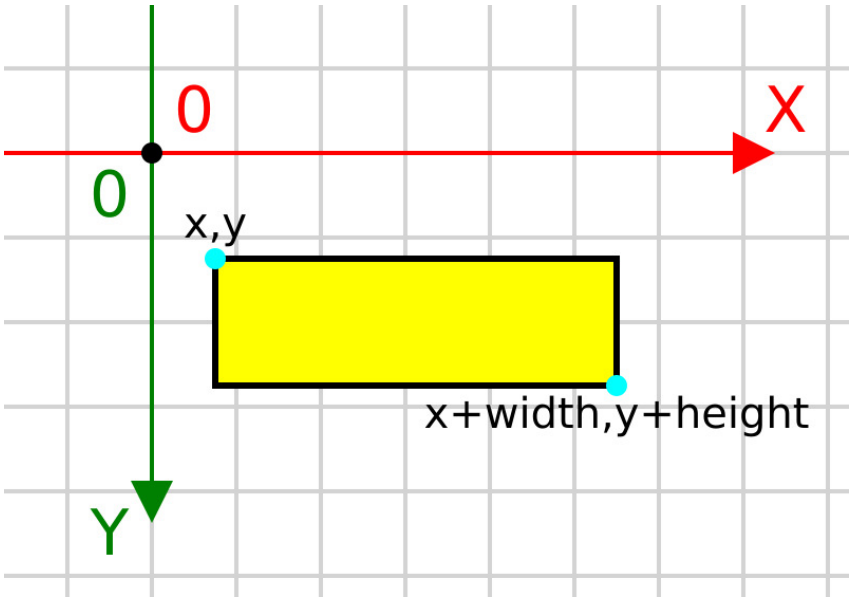


Les coordonnées peuvent être des nombres réels quelconques.

Pierre Nerzic

Primitives simples, suite

- `<rect x="x" y="y" width="width" height="height"/>` (coin haut-gauche et taille)

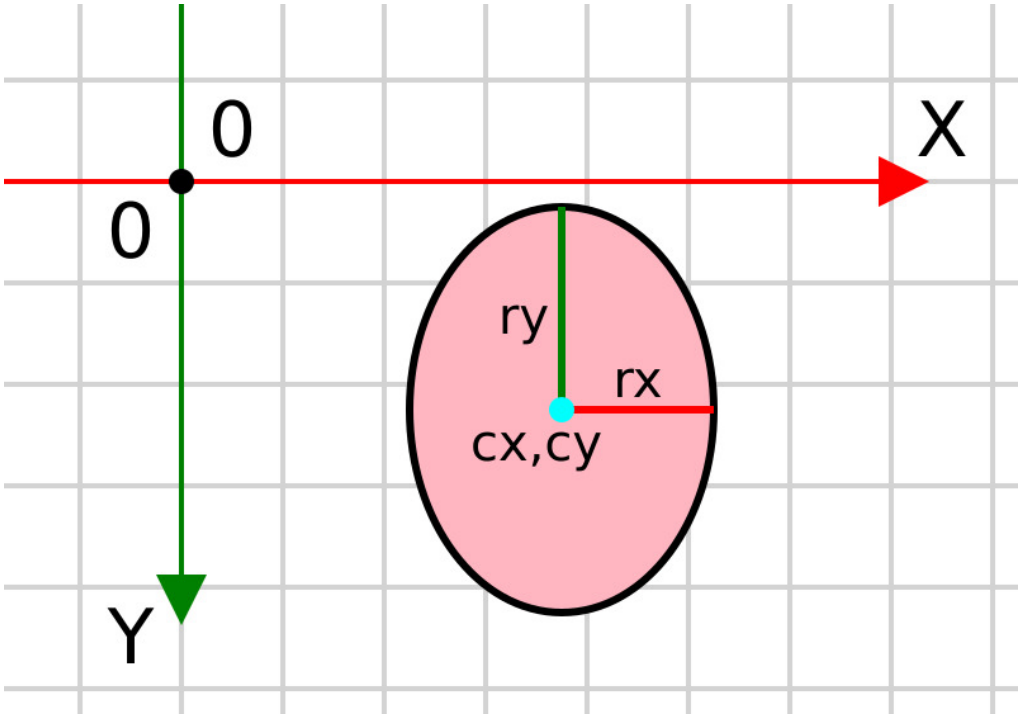


On peut utiliser des *transformations* pour changer l'orientation des figures, voir le transparent 30.

Pierre Nerzic

Primitives simples, suite

- `<ellipse cx="cx" cy="cy" rx="rx" ry="ry"/>`



- `<circle cx="cx" cy="cy" r="r"/>`

Pierre Nerzic

Modes de dessin simples

Le format SVG distingue le contour, *stroke* (« trait ») et le remplissage des figures, *fill*. On peut spécifier des couleurs différentes, au format CSS : `"nom"`, `"#RVB"`, `"#RRVVBB"`, `"rgb(r,v,b)"`, `"rgba(r,v,b,a)"`... La valeur *a* s'appelle *canal alpha*. C'est l'opacité de la couleur, de 0=transparent à 1=opaque.

Attributs pour spécifier les modes :

- contours `stroke="couleur"` ou `"none"`, `stroke-width="nb"`, `stroke-opacity="a"` (entre 0 et 1)
  - `stroke-linecap` définit si les extrémités sont arrondies `"round"` ou carrées `"square"`
  - `stroke-linejoin` définit si les jonctions sont arrondies `"round"` ou pointues `"miter"`
- remplissage `fill="couleur"` ou `"none"`, `fill-opacity="a"`

Modes de dessin simples, exemples

Les modes se mettent en tant qu'attributs des éléments de dessin :

```
<line x1="-10" y1=" 4" x2="40" y2=" 4" stroke="lightgray"
    stroke-width="0.2" stroke-opacity="0.5"/>
<circle cx="8" cy="1.8" r="1.5"
    stroke="rgb(255,0,0)" fill="#da4765"/>
<rect x="3" y="1" width="20" height="14"
    stroke="black" stroke-width="0.3" fill="none"/>
```

Le dernier élément, un rectangle, n'est pas rempli.

Pierre Nerzic

Textes

- `<text x="x" y="y">texte à écrire</text>`

La couleur du texte est définie par l'attribut `fill`. D'autres attributs permettent de définir la police, la taille, etc. :

- police comme en CSS : `font-family`, `font-size`, `font-style`
- alignement horizontal : `text-anchor="start` ou `middle` ou `end"` (doc)
- alignement vertical : `dominant-baseline="auto` ou `middle` ou `hanging"` (doc)

Styles CSS internes

Il est possible de définir des styles CSS à appliquer aux éléments.
Les propriétés sont nommées comme les attributs.

```
<svg version="1.0" xmlns="http://www.w3.org/2000/svg"...>
    <style><![CDATA[
    #arriere {
        fill: #008080;
    }
    .bulle {
        stroke: black;
        stroke-width: 0.2;
        fill: #FF4500;
    }
    ]]></style>
    <rect id="arriere" x="0" y="0" width="8" height="6"/>
    <circle class="bulle" cx="4" cy="3" r="2"/>
```


Pierre Nerzic

Styles CSS externes

Les styles CSS peuvent être hors de l'arbre SVG dans un HTML.

```
<!DOCTYPE html>
<html>
<head>
<style>
#arriere { ... }
.bulle { ... }
</style>
</head>
<body>
<svg version="1.0" xmlns="http://www.w3.org/2000/svg" ...>
    <rect id="arriere" x="0" y="0" width="8" height="6"/>
    <circle class="bulle" cx="4" cy="3" r="2"/>
</svg>
...
```

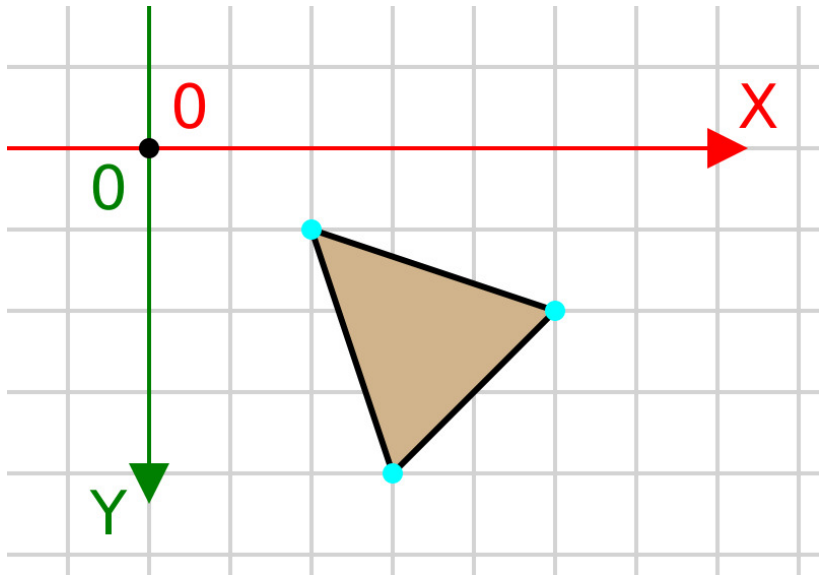

Pierre Nerzic

Chemins

L'élément <path d="chemin"/> permet de dessiner des figures complexes, incluant des lignes droites et des courbes.

```
<path
  d="M 5 2  L 3 4  L 2 1  Z"
  stroke="#000" fill="Tan"/>
```

Le chemin est défini dans l'attribut d. Ici : partir de $(5, 2)$, tracer une ligne vers $(3, 4)$ ensuite vers $(2, 1)$, enfin revenir au point de départ.



Pierre Nerzic

Chemins, suite

L'attribut `d` est assez complexe. C'est une suite de directives de tracé au format *lettre coordonnées* qui dessinent des lignes d'un point à l'autre.

- `M x y` pour démarrer un tracé en (x, y)
- `L x y` pour tracer une ligne droite du précédent point à (x, y)
- `H y` trace une ligne horizontale vers le nouveau point
- `V x` trace une ligne verticale vers le nouveau point
- `Z` pour refermer le tracé sur le premier point.

Les autres directives (lignes courbes) sont trop compliquées. On utilisera un éditeur en ligne pour dessiner des chemins complexes.

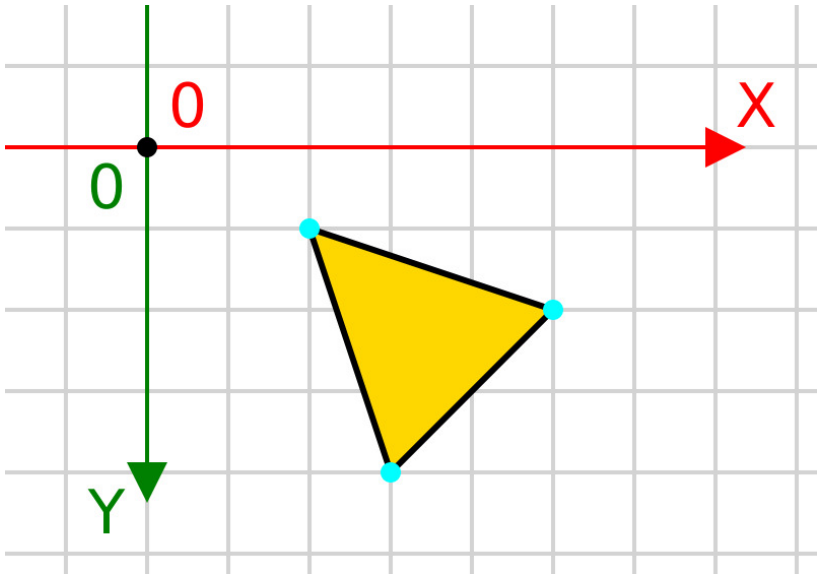
Chemins absolus et relatifs

Les directives en majuscules `L`, `H`, `V` emploient des coordonnées absolues

Les directives en minuscules emploient des coordonnées relatives au précédent point.

```
<path
  d="M 5 2  l -2 2  l -1 -3  Z"
  stroke="#000" fill="Gold"/>
```

On part de $(5, 2)$, ligne vers $(5-2, 2+2) = (3, 4)$ puis $(3-1, 4-3) = (2, 1)$, et revenir au point de départ.



Pierre Nerzic

Chemins, suite

Le fait que le chemin soit défini dans l'attribut `d` pose un problème pour l'éditer dynamiquement. On ne peut pas modifier les coordonnées d'un point facilement. Il aurait été préférable d'avoir des sous-éléments et des attributs séparés.

On fait donc appel à des bibliothèques de fonctions spécialisées dans la création et l'édition de chemins, voir d3.js dans le second CM. Le principe est qu'une fonction retourne le chemin complet, construit à partir de différents paramètres.

Par exemple, en d3.js, on peut écrire ceci :

```
svg.append("path").attr("d", d3.line(data))
```

La fonction `d3.line(data)` crée la chaîne nécessaire pour dessiner les points placés dans `data`. Voir le CM n°2.

Groupes d'objets

Il est parfois intéressant de grouper les éléments. On peut leur appliquer les mêmes modes de dessin, transformations et filtres, voir les transparents suivants. Les groupes peuvent eux-mêmes être imbriqués.

```
<g id="paysage">
    ...
    <g id="maison" transform="...">
        <rect .../>
        <path .../>
        <rect .../>
    </g>
</g>
```

Dans le groupe "maison", tous les objets enfants sont transformés de la même manière. Ils restent donc attachés ensemble.

Pierre Nerzic

Transformations

Elles sont utiles pour décaler, pivoter, agrandir/rétrécir les éléments d'une image. Les transformations sont définies soit par une composition de modifications, soit par une matrice.

Les compositions de modifications sont plus simples. On les place dans un attribut `transform="..."`. Ex:
`transform="translate(10 10) rotate(45)"`

- `translate(dx dy)` (sans virgule entre dx et dy)
  NB: dy positif fait descendre, car l'axe y va vers le bas.
- `rotate(angle)` l'angle est en degrés, par rapport au repère, attention: à cause de la direction de l'axe y, les rotations semblent dans le sens horaire, en réalité le sens est trigonométrique dans le repère.
- `scale(k)` ou `scale(kx ky)`

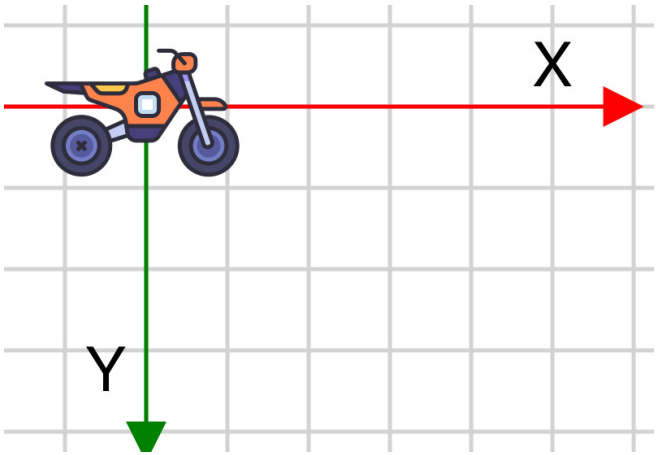
Pierre Nerzic

Composition des transformations

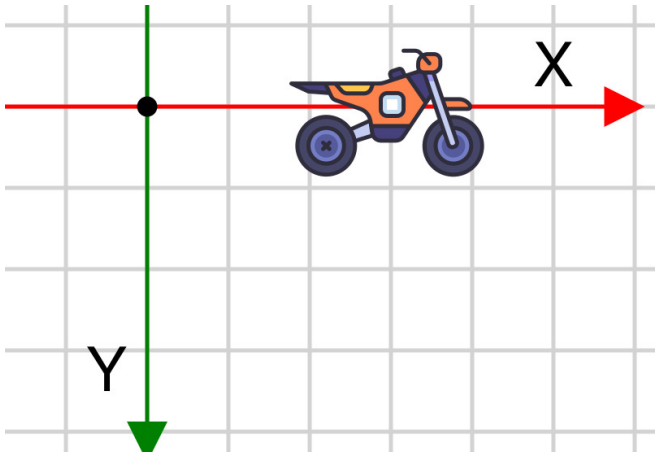
Le principe est celui de la composition de fonctions en mathématiques. Quand on écrit $f_1(f_2(f_3(x)))$, on calcule d'abord $f_3(x)$, puis on applique f_2 au résultat, puis enfin f_1. On l'écrit $f1 \circ f2 \circ f3$ en mathématiques, et simplement $f_1 \quad f_2 \quad f_3$ en SVG.

Pour comprendre, d'abord une simple translation :

Au repos, en (0,0)



translate(3 0)

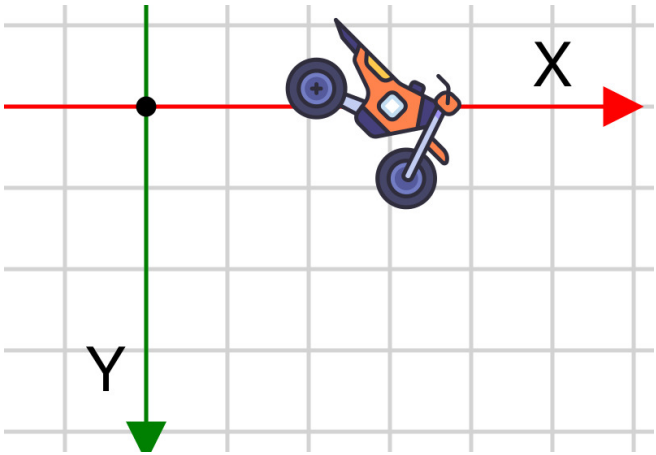


Pierre Nerzic

Ordre des transformations

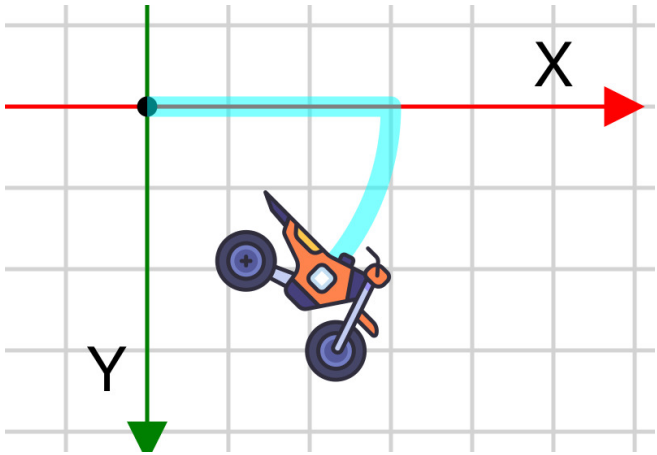
On ajoute une rotation. L'ordre d'application des modifications n'est pas commutatif. C'est l'inverse de l'ordre d'écriture.

translate(3 0) rotate(45)



La moto part du centre $(0, 0)$, elle pivote de 45°, puis est décalée en $(3, 0)$.

rotate(45) translate(3 0)



La moto est décalée en $(3, 0)$ puis il y a une rotation globale de 45° autour de $(0, 0)$.

Pierre Nerzic

Centre des transformations

Par défaut, les transformations `rotate` et `scale` sont par rapport à $(0, 0)$, le centre des coordonnées globales.

On peut changer la référence des rotations et homothéties en ajoutant l'attribut `transform-origin="x0 y0"` à l'élément concerné.

```
<g id="objet"
    transform-origin="2 3"
    transform="rotate(45)">
    ...
</g>
```

On fait pivoter le groupe de 45° autour de (2,3).

Des groupes imbriqués héritent des transformations des parents et peuvent définir leur propre centre de transformations.

Pierre Nerzic

Filtres et dégradés

La norme SVG permet d'appliquer des filtres sur un dessin et de colorer les surfaces avec des dégradés (*gradients*) et des motifs (*patterns*), mais la complexité dépasse ce qui est utile pour ce cours.

L'objectif du cours est d'apprendre à dessiner des diagrammes dynamiquement, par programmation, et non pas de réaliser des dessins esthétiques.

Donc nous passons à l'étude aux aspects programmation.

Programmation JavaScript

Principes

L'idée générale est de créer et modifier les balises SVG à l'aide de fonctions JavaScript. De plus, les dessins dépendent de données dynamiques, par exemple obtenues avec AJAX (voir R5.C.04).

On utilisera les méthodes suivantes :

- `document.querySelectorAll(selector)` et `document.getElementById(id)`
- `document.createElementNS(ns, name)`
- `element.appendChild(child)`
- `element.setAttribute(name, value)`

Voyons comment les employer dans une page HTML5.

Cadre général

Il faut mettre en place :

- un élément `<svg id="dessin" xmlns=.../>` vide ou partiellement rempli
- un script exécuté au chargement de la page

```
<body>
  <svg id="dessin" width="..." height="..."  ...></svg>

  <script>
    // obtenir l'élément <svg>
    const svg = document.getElementById("dessin")
    // créer/modifier le contenu de l'élément svg
    ...
  </script>
</body>
```


Pierre Nerzic

Création d'éléments

Voici une fonction pratique pour les créer dans le bon *namespace* :

```
const NS_SVG = 'http://www.w3.org/2000/svg'

function appendElement(parent, name, attributes={}) {
    // créer l'élément et l'ajouter au parent
    const element = document.createElementNS(NS_SVG, name)
    parent.appendChild(element)
    // affecter ses attributs
    for (const [attr, value] of Object.entries(attributes)) {
      element.setAttribute(attr, value)
    }
    // retourner l'élément créé
    return element
}
```


Pierre Nerzic

Modification d'un attribut d'élément

Il faut d'abord récupérer cet élément dans une variable. Il faut soit l'avoir gardé dans une variable à sa création, soit arriver à le désigner par un sélecteur CSS, voir la doc de référence.

Voici un exemple de fonction :

```
function setSelectedElementAttribute(selector, attr, value) {
    // récupérer l'élément
    const nodes = document.querySelectorAll(selector)
    if (nodes.length != 1) {
      throw `element identified by "${selector}" is not unique`
    }
    const element = nodes[0]
    // changer ou définir l'attribut
    element.setAttribute(attr, value)
}
```

Suppression d'un élément

Il faut seulement utiliser la méthode `element.remove()`.

Pierre Nerzic

Proposition pour les TP

En TD et TP, il sera proposé de construire une classe facilitant la gestion des SVG. Voici quelques unes de ses méthodes publiques :

```
class SVGelement {
  static fromSelector(selector: string)
  appendElement(name: string, attributes={}): SVGelement
  prependElement(name: string, attributes={}): SVGelement
  setAttributes(attributes: {}): SVGelement
}
```

L'astuce est que toutes ces méthodes retournent des `SVGelement` qui sont également des instances d'éléments du DOM, et qui possèdent donc tous ces méthodes. C'est grâce à JavaScript qui permet d'ajouter des méthodes à n'importe quel objet.

Application

Voici un exemple de mise en pratique :

```
class Voiture {
  constructor(parent, taille=1) {
    this.g = parent.appendElement("g")
    this.roueAR = this.g.appendElement("circle", {
        cx: taille*0.75,
        cy: 0,
        r: taille*0.2,
        fill: "black",
    })
  }
}
const svg = SVGelement.fromSelector("#dessin")
const voiture = new Voiture(svg, 10)
```


Pierre Nerzic

Bibliothèque GSAP

Présentation

GSAP, GreenSock Animation Platform peut être employée dans tout document HTML5 pour animer n'importe quelle propriété : éléments, CSS, SVG, etc. L'intérêt est de fonctionner de manière uniforme sur tous les navigateurs, permettant d'éviter des directives de compatibilité (`-moz-*`, `-webkit-*`, etc).

C'est une bibliothèque JavaScript qui se présente, une fois chargée, sous la forme d'un objet, `gsap` possédant un petit nombre de méthodes, comme `to` et `timeline`. Ce sont des patrons *fabriques* qui retournent un objet *tween* représentant une animation, c'est à dire une interpolation entre un état initial/actuel et un état final.

Les états initial et final sont définis par des valeurs d'attributs ou des propriétés CSS du document HTML.

Remarque importante

GSAP évolue régulièrement. Il en est à la version 3 qui est décrite par cette documentation.

Le problème, qu'on retrouve avec d'autres API comme d3.js, c'est qu'il y a de très nombreux tutoriels et forums concernant des versions antérieures appelées *TweenLite* et *TweenMax*. Il faut faire attention à ne pas reprendre ces bouts de code directement.

Cette page explique très bien comment migrer de *TweenLite* et *TweenMax* vers gsap3.

Un exemple complet

Dessiner un disque animé (balle.htm) :

```
<html>
<head>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/gsap/3.12.
</head>
<body>
  <svg width="200" height="200" viewBox="0 0 100 100">
    <line x1="0" y1="99" x2="100" y2="99" stroke="black"/>
    <circle id="balle" cx="50" cy="75" r="25" fill="red"/>
  </svg>
  <script>
gsap.to("#balle", {cy:25, duration:0.5, yoyo:true, repeat:-1})
  </script>
</body>
</html>
```


Pierre Nerzic

Analyse de l'exemple

Soit un objet du DOM à animer, ex: un élément `<circle>` d'un SVG. Animer = changer la valeur d'un attribut, ex: la position `cy`.

La méthode `gsap.to(sélecteur, paramètres)` déclenche l'animation de l'élément sélectionné :

- le sélecteur désigne l'élément à animer :
  - une chaîne écrite comme en CSS (tuto et doc des sélecteurs), ex : `"#identifiant"`, `"élément>.classe"`, etc.
  - ou directement un élément du DOM
  - ça peut aussi être une liste de sélecteurs ou d'éléments
- Les paramètres sous forme d'un objet `{attr: valeur,...}` :
  - les nouvelles valeurs d'attributs (valeurs à atteindre à la fin de l'animation)
  - le paramétrage de l'animation (transparents suivants).

Paramètres importants

Petite sélection de ce qui existe, voir la doc :

- `duration: D` durée en nombre de secondes (D=réel>0)
- `delay: D` nombre de secondes avant de commencer (D=réel≥ 0)
- `repeat: N` nombre de répétitions *supplémentaires* (N+1 en tout), -1 = infini
- `yoyo: true` pour faire un aller-retour, `false` par défaut, `repeat` doit être au moins 1,

```
gsap.to("#balle", {
    cy: 25,             // attribut à animer
    duration: 0.5,      // paramètres de l'animation
    yoyo: true,
    repeat: -1
})
```

Paramètres importants, suite

- `ease`: au format *"fonction.bords"*, p. ex. `"sine.inOut"` spécifie la dérivée du mouvement : accélérations et freinages.
  - Il y a de nombreuses valeurs pour le mot clé *fonction*, voir la doc (avec un diagramme interactif pour voir l'effet) :
    - `none` donne une vitesse constante,
    - `power1`..`power4`, `sine` ralentissent vers les bords,
    - `back`, `elastic`, `bounce` dépassent la limite puis revient lentement en arrière,
    - Certaines fonctions sont paramétrables.
  - le mot clé *bords* indique quelles limites sont concernées `in` (début d'animation), `inOut` (début et fin) ou `out` (fin)

Animation des SVG

GSAP permet d'animer tout élément d'un document HTML, doc : `<div>` et autres, et également les images SVG incluses.

Il y a des raccourcis pour animer les transformations (doc) :

- translations : `x:N` et `y:N` à préférer à `left`, `top` et `margin`
- rotation : `rotation:N`
- taille : `scale:N`

```
gsap.to("#balle", {
    y: -50,           // translation sur la position initiale
    scale: 1.2,
    duration: 0.5, ...
})
```

En interne, ces transformations sont réalisées par une matrice.

Remarques sur l'animation

Il faut savoir que, par défaut, GSAP altère le style de l'élément concerné et non pas ses attributs.

C'est à dire que dans l'exemple précédent, au lieu de modifier `cy="75"`, GSAP ajoute un attribut `style="cy:`*N*`px;"`. Donc, au cours de l'animation, l'élément se trouve ainsi :

```
<circle id="balle" cx="50" cy="75" r="25" fill="red"
    style="cy: 43px;"/>
```

Dans certains cas spécifiques, il faut vraiment modifier l'attribut et on le fait de cette façon, en encapsulant les attributs à modifier directement dans un objet `attr` :

```
gsap.to("#balle", {
    attr: {cy: 25},      // attribut à animer directement
    ...
```


Pierre Nerzic

Assemblage d'animations (méthode 1)

On peut enchaîner plusieurs animations à l'aide d'une propriété appelée `onComplete`. On doit fournir le nom d'une fonction ou une lambda qui est exécutée à l'issue de l'animation :

```
gsap.to("#balle", {
    y: -50,          // translation sur la position initiale
    duration: 0.5,
    onComplete: () => {
        gsap.to("#balle", {
            r: 50,
            duration: 0.5,
        })
    },
})
```

À noter que c'est plutôt une sorte d'astuce.

Assemblage d'animations (méthode 2)

GSAP permet d'assembler plusieurs animations sous la forme d'une chronologie (*timeline*). Ça permet de *monter* différentes animations, comme des clips dans un film : on démarre par telle animation, puis on enchaîne avec telle autre... doc explicative

D'abord on crée un objet *timeline* puis on l'utilise pour créer des animations :

```
let tl = gsap.timeline()
tl.to("#balle1", {cy: 25, duration: 0.5, yoyo:true, repeat: 1})
tl.to("#balle2", {cy: 25, duration: 0.5, yoyo:true, repeat: 1})
```

Les deux animations sont mises bout à bout. Les balles bougent successivement.

Position d'une animation dans la chronologie

Il est possible de placer les animations comme on veut dans la chronologie, par exemple les faire chevaucher. C'est avec un paramètre supplémentaite, appelé *position*, voir doc.

```
let tl = gsap.timeline()
tl.to("#balle1", {cy: 25, ...}, 1)
tl.to("#balle2", {cy: 25, ...}, "<+0.2")
```

La première balle bouge après 1 seconde d'attente, et la deuxième balle démarre 0.2s après. Il y a de très nombreuses possibilités.

Pierre Nerzic

Imbrication des chonologies

Les *timelines* peuvent être imbriquées, et être positionnée comme de simples animations, à l'aide de la méthode add (doc) :

```
let tl1 = gsap.timeline()
tl1.to("#balle1", {...})
tl1.to("#balle1", {...})

let tl2 = gsap.timeline()
tl2.to("#balle2", {...})
tl2.to("#balle2", {...})

let tl = gsap.timeline()
tl.add(tl1)
tl.add(tl2, "<")    // démarre en même temps que tl1
```

Petit pb : empêcher que tl1 démarre quand on la définit, voir TD2.

Pierre Nerzic

Méthodes de GSAP

GSAP propose :

- `gsap.to(sélecteur, paramètres)` pour définir la situation finale à partir de la situation actuelle
- `gsap.fromTo(sélecteur, paramsDépart, paramsFin)` pour définir les situations initiale et finale
- `gsap.set(sélecteur, paramètres)` pour affecter ces paramètres immédiatement, sans animation

Pierre Nerzic

Appels réguliers à une fonction

La méthode `gsap.ticker.add` (doc) permet d'appeler une fonction, ou méthode, aussi souvent que possible :

```
function display(time) {
    ...
}

gsap.ticker.add(display)
```

La fonction `display` peut, p. ex., modifier un ensemble complexe d'attributs dans un dessin SVG directement en fonction du temps écoulé.

NB: `gsap.to` ne permet que d'interpoler des attributs entre une valeur initiale et une valeur finale.

Pierre Nerzic

gsap.ticker.add vs setInterval

L'API DOM propose une fonction similaire : `setInterval` (doc) :

```
function display() {
    ...
}

setInterval(display, 250) // appel à display() 4x par seconde
```

- `setInterval` appelle une fonction régulièrement, peu importe la charge du navigateur
- `gsap.ticker.add` appelle une fonction aussi souvent que possible, selon la charge du navigateur

Donc `gsap.ticker.add` garantit une animation fluide partout, mais pas régulière.

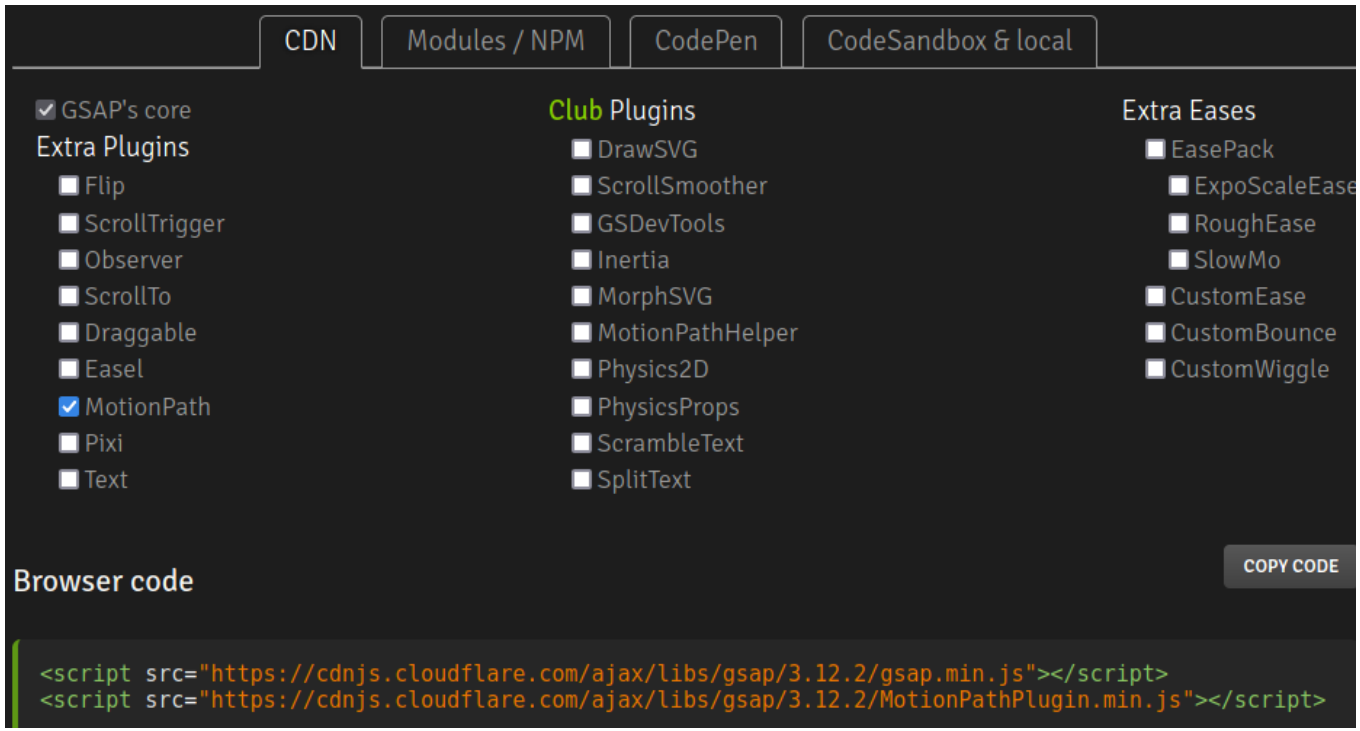
Extensions de GSAP

GSAP propose de nombreuses extensions, certaines gratuites, d'autres payantes, pour des animations complexes.

- gestion du défilement : ScrollTrigger, ScrollToPlugin
- physique 2D : Physics2DPlugin, PhysicsPropsPlugin, InertiaPlugin
- suivi de chemins : MotionPathPlugin, BezierPlugin
- transformation de chemins : MorphSVGPlugin, DrawSVGPlugin
- animation CSS : CSSRulePlugin, CSSPlugin

Installation des extensions

Utiliser ce formulaire pour obtenir les inclusions CDN pour les scripts nécessaires. La colonne de gauche *Extra Plugins* donne la liste des extensions gratuites, les autres sont payantes.



Suivi de chemins avec MotionPathPlugin

Il déplace un élément le long d'un chemin SVG quelconque.

```
<svg ...>
<path id="chemin" d="M 17 62 c..."/>
<circle id="balle" cx="0" cy="0" r="25" fill="red"/>
</svg>
<script>
gsap.to("#balle", {
    motionPath: {        // config de MotionPathPlugin
      path: "#chemin",   // sélecteur ou élément
    },
    duration: 10,        // config standard de gsap.to
})
```

Des options (doc) permettent de décaler (`align`, `alignOrigin`) et orienter l'objet (`autoRotate`) par rapport au chemin.

Vitesse de déplacement

Il manque de quoi paramétrer la vitesse de déplacement le long du chemin. On ne peut, a priori, que définir le temps de déplacement. Il suffit de calculer la longueur du chemin et la diviser par la vitesse voulue pour obtenir le temps de déplacement.

```
const chemin = document.getElementById("chemin")
const lng = chemin.getTotalLength()
const vitesse = 100

gsap.to("#balle", {
    motionPath: {
      path: chemin,
    },
    duration: lng / vitesse,
})
```

C'est tout pour aujourd'hui

Cette présentation est finie. Rendez-vous en TD et TP pour mettre cela en pratique.

Le prochain cours présentera d3.js

Pierre Nerzic