

R5.A.06 - Prog multimédia

Pierre Nerzic - pierre.nerzic@univ-rennes1.fr

septembre 2024

Abstract

Il s'agit des transparents du cours mis sous une forme plus facilement imprimable et lisible. Ces documents ne sont pas totalement libres de droits. Ce sont des supports de cours mis à votre disposition pour vos études sous la licence *Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International*.



Version du 18/11/2024 à 08:51

Table des matières

1		5
1.1	Introduction	5
1.1.1	Programme national	5
1.1.2	Visualisation de données	5
1.1.3	Concepts du cours	5
1.1.4	Plan simplifié de cet enseignement	6
1.2	Images vectorielles SVG	6
1.2.1	Deux sortes d'images	6
1.2.2	Utilité des images vectorielles	6
1.2.3	Redimensionnement d'images	7
1.2.4	Structure générale d'images SVG	9
1.2.5	Taille de l'image	10
1.2.6	Coordonnées	10
1.2.7	Définition de la zone visible	10
1.2.8	Exemple de <i>viewBox</i> (zone visible)	11

1.2.9	Primitives simples	11
1.2.10	Modes de dessin simples	13
1.2.11	Modes de dessin simples, exemples	14
1.2.12	Textes	14
1.2.13	Styles CSS internes	14
1.2.14	Styles CSS externes	15
1.2.15	Chemins	15
1.2.16	Chemins absolus et relatifs	16
1.2.17	Groupes d'objets	17
1.2.18	Transformations	18
1.2.19	Composition des transformations	18
1.2.20	Ordre des transformations	19
1.2.21	Centre des transformations	21
1.2.22	Filtres et dégradés	21
1.3	Programmation JavaScript	21
1.3.1	Principes	21
1.3.2	Cadre général	21
1.3.3	Création d'éléments	22
1.3.4	Modification d'un attribut d'élément	22
1.3.5	Suppression d'un élément	23
1.3.6	Proposition pour les TP	23
1.3.7	Application	23
1.4	Bibliothèque GSAP	23
1.4.1	Présentation	23
1.4.2	Remarque importante : évolutions de GSAP	24
1.4.3	Un exemple complet	24
1.4.4	Analyse de l'exemple	24
1.4.5	Concepts de GSAP	25
1.4.6	Paramètres importants	25
1.4.7	Animation des SVG	26
1.4.8	Remarques sur l'animation	26
1.4.9	Assemblage d'animations (méthode 1)	26
1.4.10	Assemblage d'animations (méthode 2)	27
1.4.11	Position d'une animation dans la chronologie	27

1.4.12	Imbrication des chronologies	27
1.4.13	Méthodes de GSAP	28
1.4.14	Appels réguliers à une fonction	28
1.4.15	<code>gsap.ticker.add</code> vs <code>setInterval</code>	28
1.4.16	Extensions de GSAP	28
1.4.17	Installation des extensions	29
1.4.18	Exemple : suivi de chemins avec le plugin <code>MotionPath</code>	29
1.4.19	Vitesse de déplacement	30
1.4.20	C'est tout pour aujourd'hui	30
2	d3.js	31
2.1	Introduction	31
2.1.1	Concepts généraux	31
2.1.2	Exemples à aller voir	31
2.1.3	Concepts de d3.js	32
2.1.4	Points forts, points faibles	32
2.2	Mécanismes de d3.js	32
2.2.1	Installation dans une page HTML	32
2.2.2	Sélection d'éléments	33
2.2.3	Parent d'une sélection	33
2.2.4	Modifications des éléments sélectionnés	33
2.2.5	Convention d'indentation	34
2.2.6	Valeurs fournies aux méthodes	34
2.2.7	Création/modification/suppression automatique d'éléments	35
2.2.8	Méthode <code>data()</code> approfondie	35
2.2.9	Sélections <i>enter</i> et <i>exit</i>	36
2.2.10	Liaison des données aux éléments du DOM	36
2.2.11	Méthode <code>join()</code> approfondie	37
2.3	Création d'un graphique avec d3.js	37
2.3.1	Présentation	37
2.3.2	Cadre général	38
2.3.3	Objectif	38
2.3.4	Réalisation	39
2.3.5	Résultat	39

2.4	Compléments graphiques	39
2.4.1	Introduction	39
2.4.2	Mise à l'échelle	40
2.4.3	Axes et graduations	41
2.4.4	Convention des marges	42
2.4.5	Ajout d'axes	43
2.4.6	Méthode <code>call</code>	43
2.4.7	Ajout d'un titre	44
2.5	Données dynamiques	44
2.5.1	Mise à jour des données	44
2.5.2	Obtention des données	45
2.5.3	Transitions	45
2.6	Interactions avec l'utilisateur	46
2.6.1	Écouteurs sur les éléments	46
2.6.2	Zoom et panoramique à la souris	46
2.6.3	Zoom sur les axes	46
2.7	Techniques avancées	47
2.7.1	Une fonction est un objet	47
2.7.2	Fonctions internes	47
2.7.3	Fermetures (<i>closure</i>)	47
2.7.4	Accesseurs et modificateurs de fermetures	48
2.7.5	Fermetures dans <code>d3.js</code>	48
2.7.6	Ajout de méthodes pour une classe	49
2.7.7	Ajout si absent	49

Semaine 1

1.1. Introduction

1.1.1. 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 chacun 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 la *data visualization*.

1.1.2. 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.

1.1.3. 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.

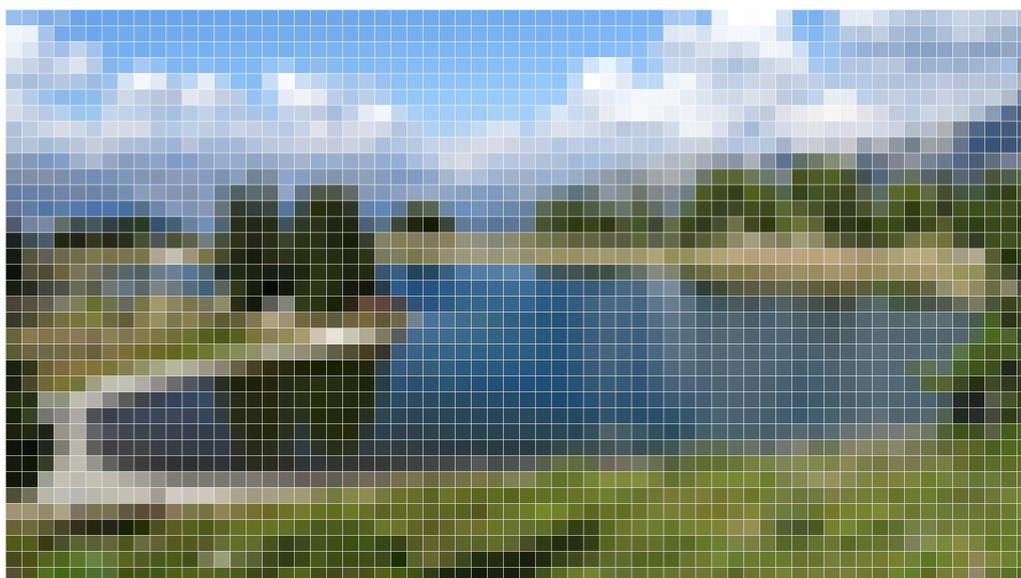


Figure 1: image matricielle

1.1.4. 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: ce cours est le début de R5.C.04 pour le parcours C.

Attention, l'évaluation se fait sur les deux TP, mais si ça se passe mal, ça sera un devoir sur table.

1.2. Images vectorielles SVG

1.2.1. Deux sortes d'images

Les ordinateurs gèrent deux sortes d'images :

- Images matricielles (*raster* ou *bitmap*) : elles sont définies par un tableau rectangulaire de pixels
- Images vectorielles : elles sont définies par des figures géométriques : points, lignes, rectangles, ellipses, etc. portant des propriétés ex: position, dimensions, épaisseur, couleur, transparence...

1.2.2. Utilité des images vectorielles

Applications :



Figure 2: image vectorielle



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

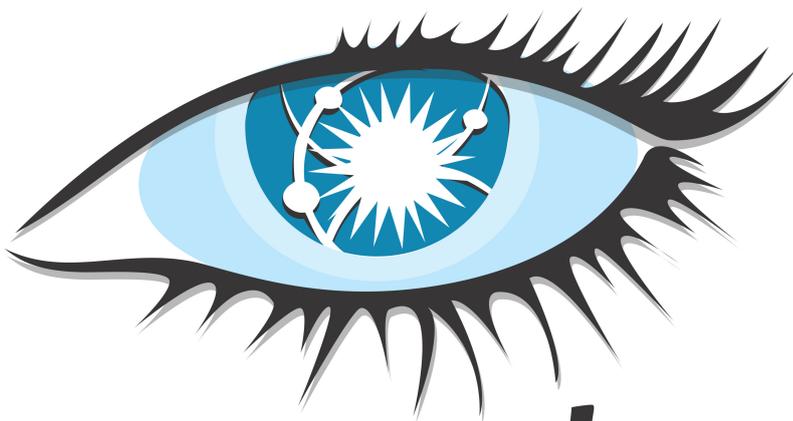
parce que :

- 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

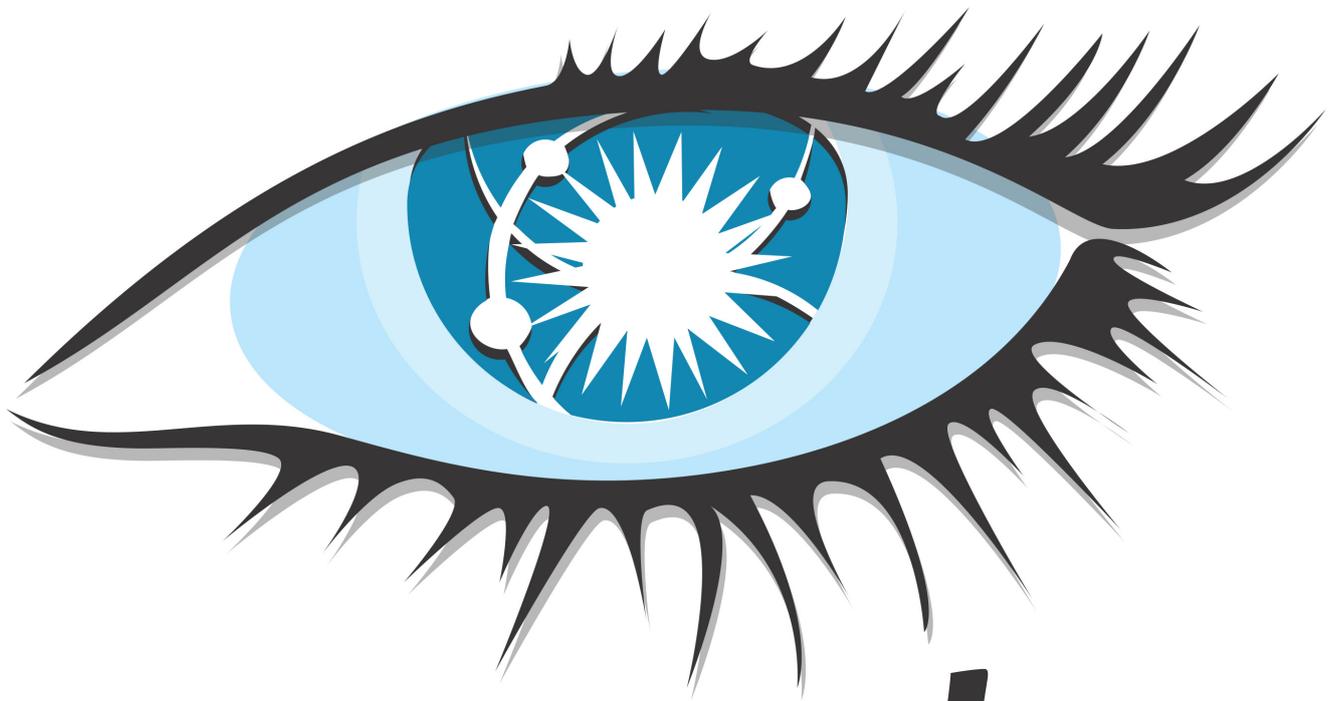
1.2.3. Redimensionnement d'images

Image vectorielle





cassandra



cassandra

Image matricielle





1.2.4. 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 filles de `<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.

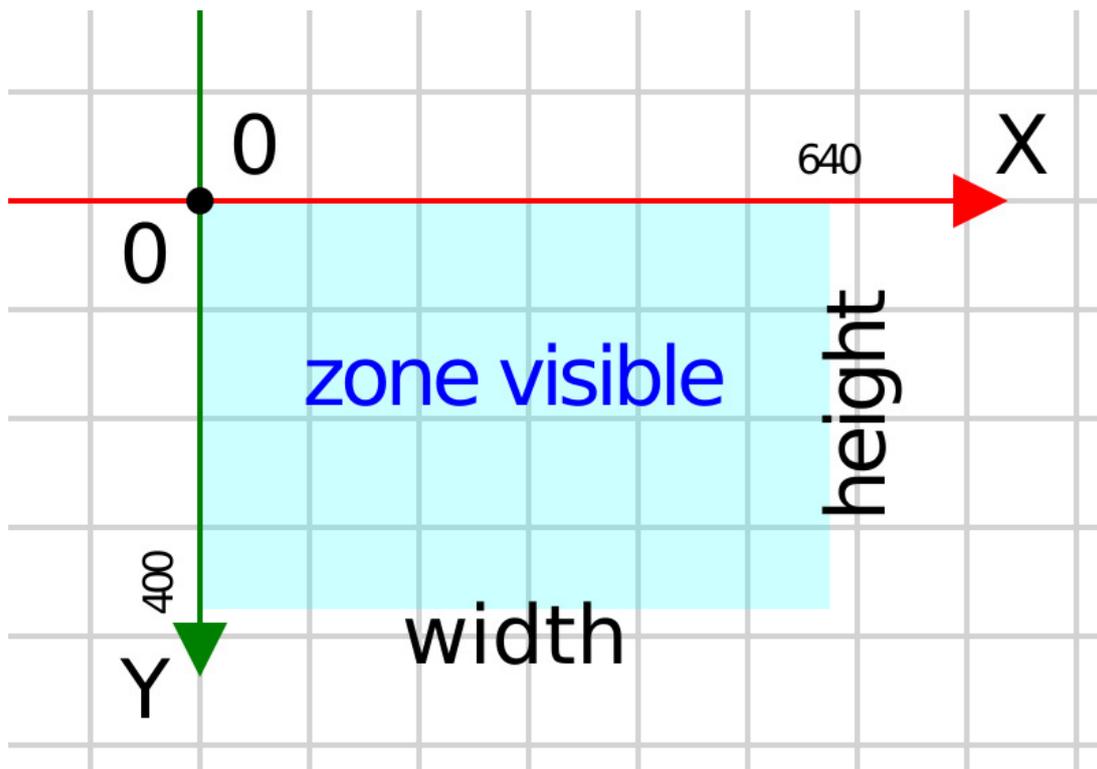


Figure 3: Coordonnées

1.2.5. 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">
...

```

Ces attributs définissent les limites de la zone visible, c'est-à-dire les coordonnées utilisables. Ce sont des *réels* qui vont de 0 à `width` pour `x`, et de 0 à `height` pour `y`.

S'ils sont absents dans un document HTML, alors la taille est entièrement déterminée par le conteneur de l'image.

1.2.6. Coordonnées

1.2.7. 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 de dimensions.

```
<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) .

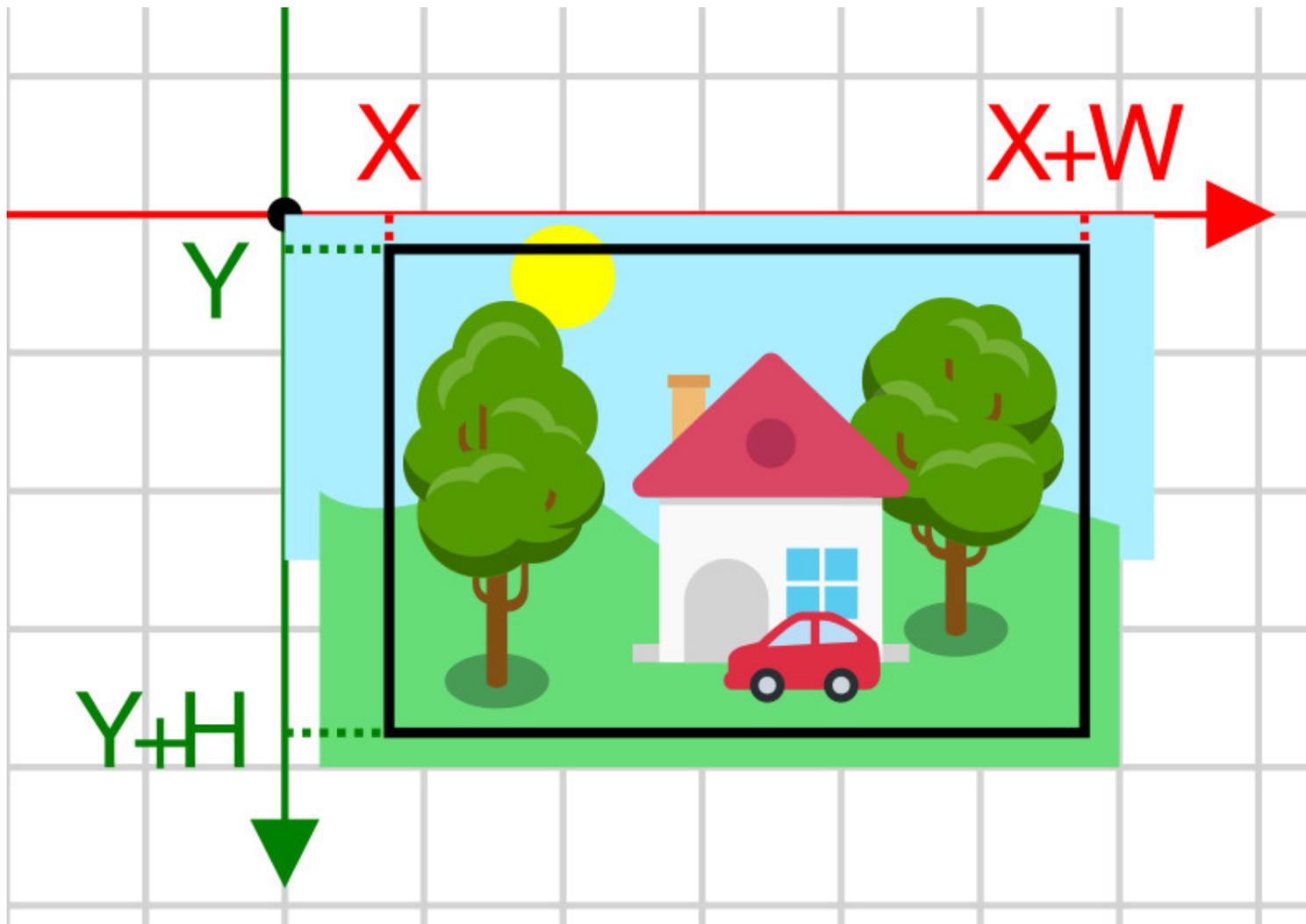


Figure 4: Coordonnées

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 $k = 4$

1.2.8. 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 :

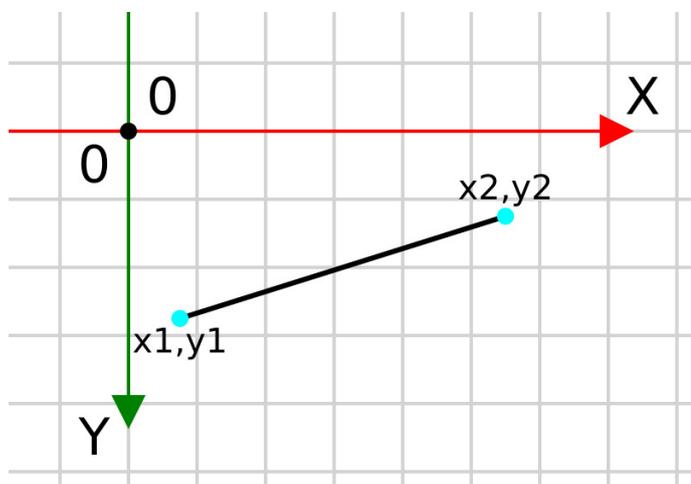
1.2.9. Primitives simples

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"/>`

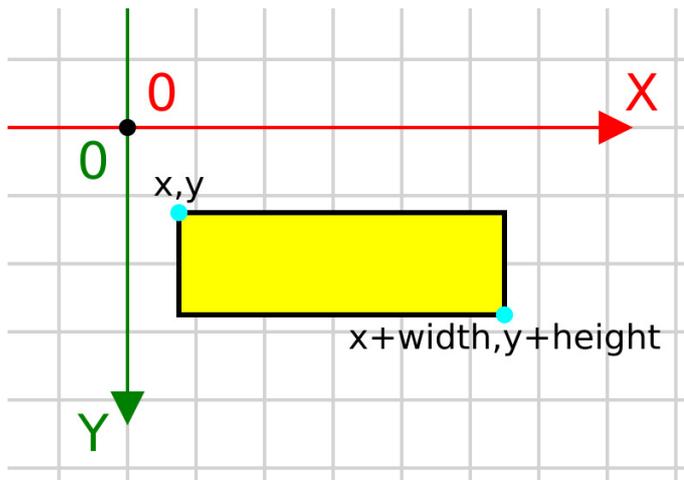


Figure 5: Coordonnées



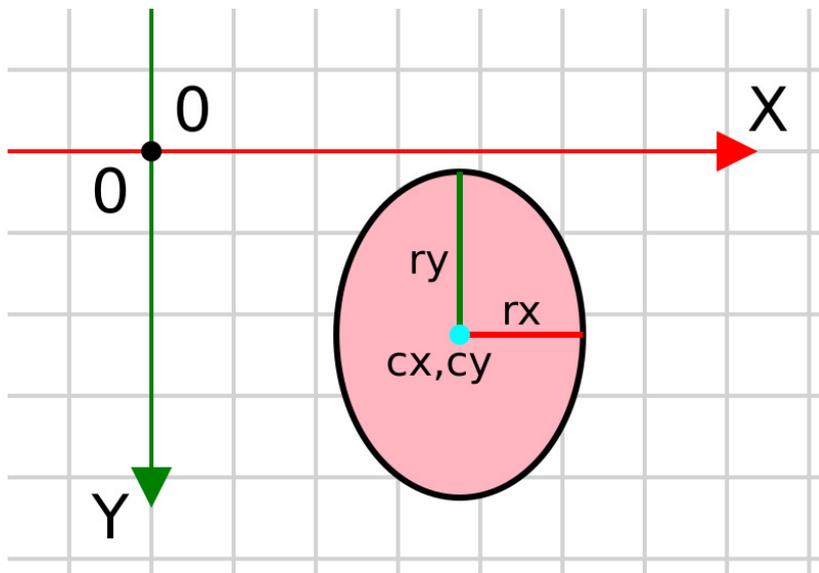
Les coordonnées peuvent être des nombres réels quelconques.

- `<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 18.

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



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

1.2.10. 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"`

1.2.11. 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.

1.2.12. 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](#))

1.2.13. 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"/>
```

1.2.14. Styles CSS externes

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

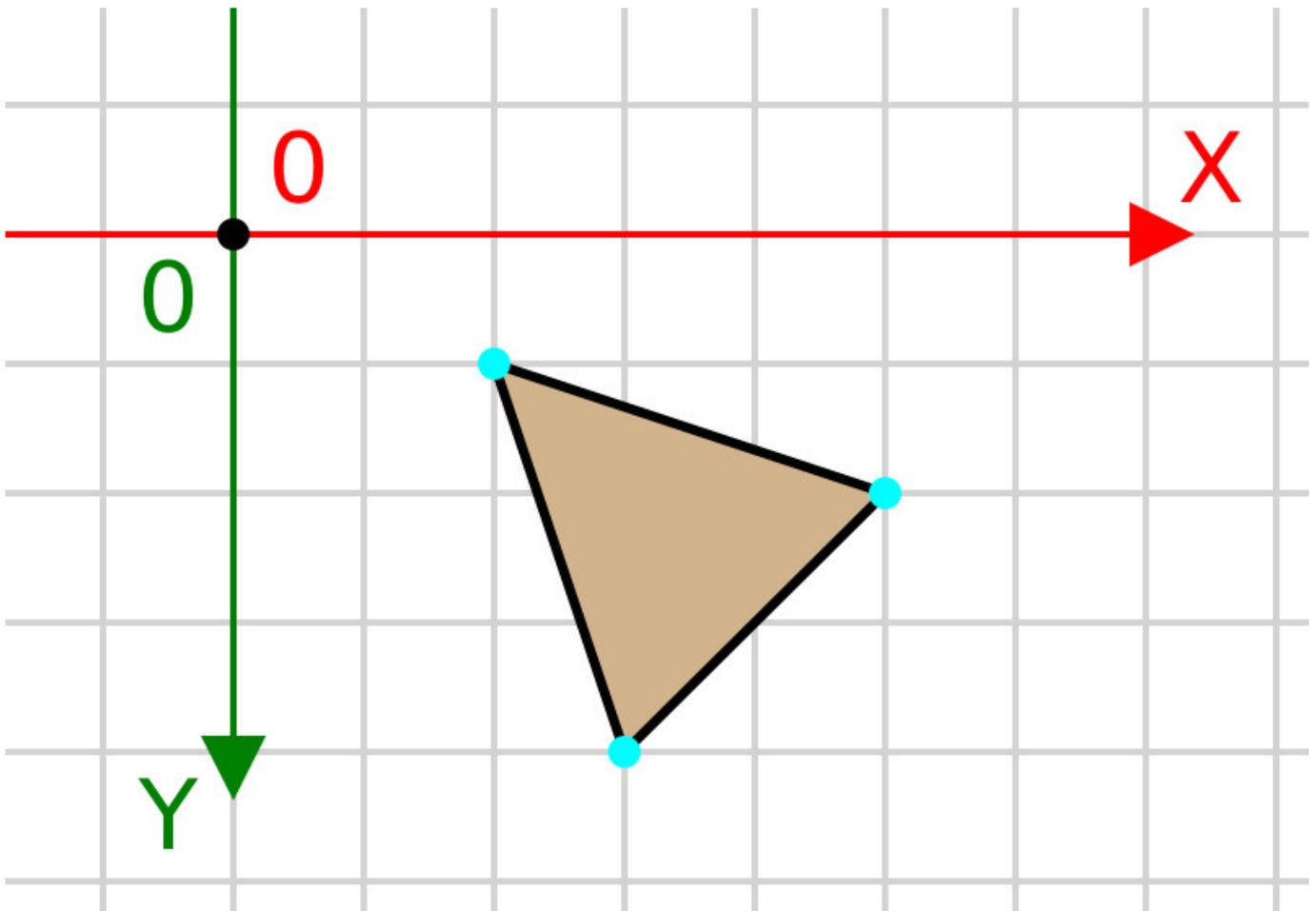
```
<!DOCTYPE html>
<html>
<head>
<style>
##arriere { ... }
.bulle { ... }
</style>
</head>
<body>
<svg version="1.0" xmlns="http://www.w3.org/2000/svg" ...>
  <rect id="arriere" x="0" y="0" width="8" height="6"/>
  <circle class="bulle" cx="4" cy="3" r="2"/>
</svg>
...
```

1.2.15. 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.



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.

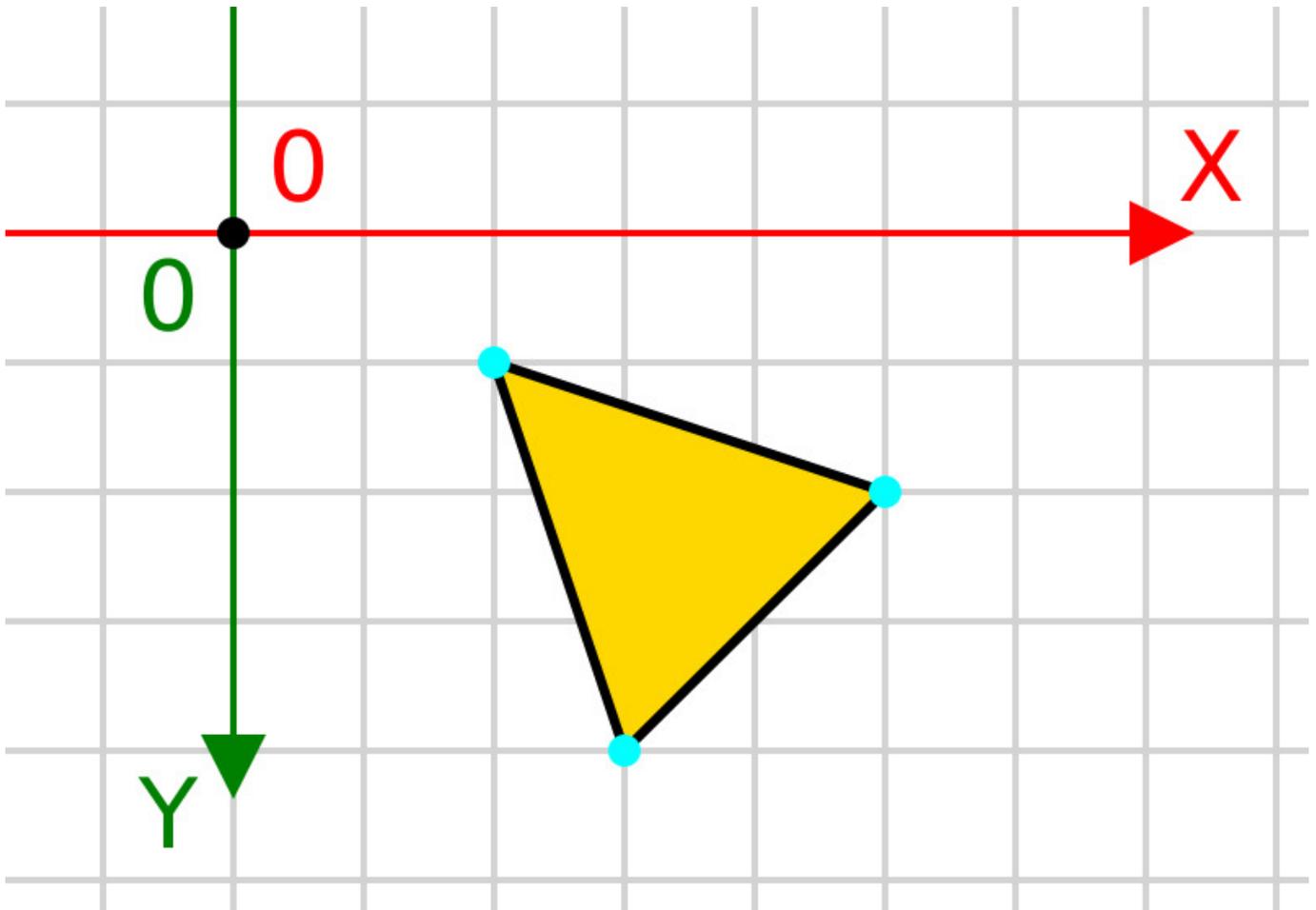
1.2.16. Chemins absolus et relatifs

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 1 -2 2 1 -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.



Le fait que le chemin soit défini dans l'attribut `d` pose un problème pour l'éditer dynamiquement. On ne peut pas facilement modifier les coordonnées d'un point. 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.

1.2.17. 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.

1.2.18. 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)"`

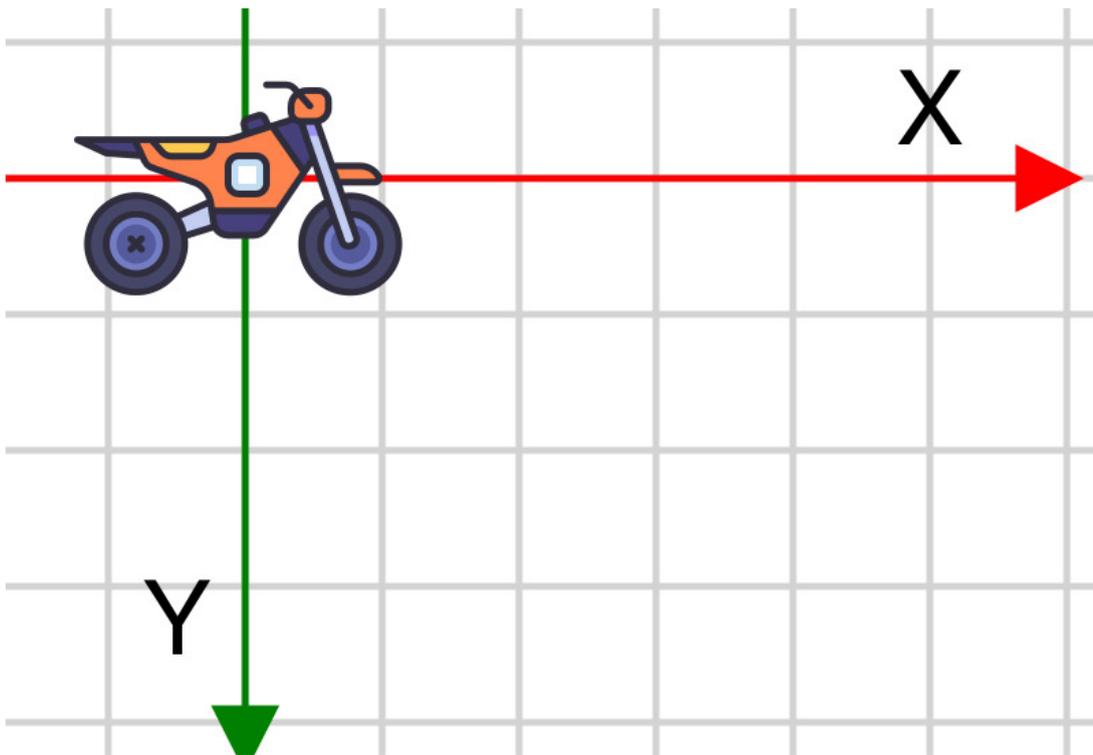
- `translate(dx dy)` (sans virgule entre dx et dy)
NB: dy positif fait descendre, car l’axe y va vers le bas.
- `rotate(angle)` l’angle est en degrés, par rapport au repère, attention: à cause de la direction de l’axe y , les rotations semblent dans le sens horaire, en réalité le sens est trigonométrique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

1.2.19. Composition des transformations

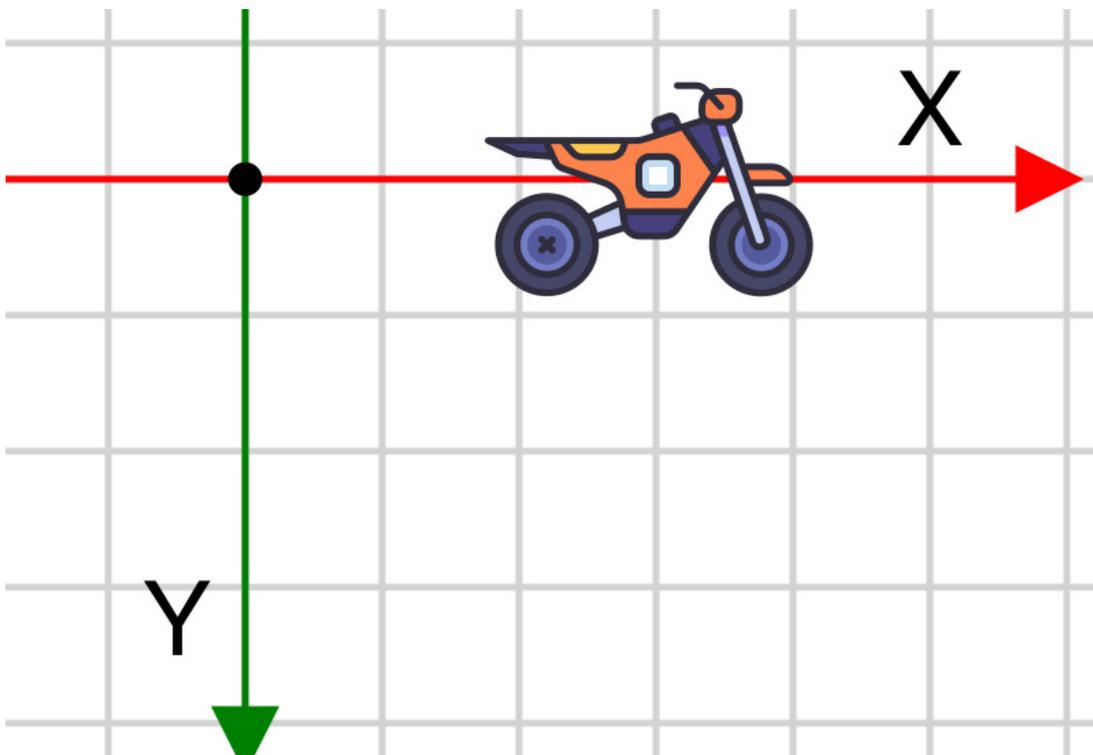
Le principe est celui de la composition de fonctions en mathématiques. Quand on écrit $f_1(f_2(f_3(x)))$, on calcule d’abord $f_3(x)$, puis on applique f_2 au résultat, puis enfin f_1 . On l’écrit $f_1 \circ f_2 \circ f_3$ en mathématiques, et simplement $f_1 f_2 f_3$ en SVG.

Pour comprendre, d’abord une simple translation :

Au repos, en $(0,0)$



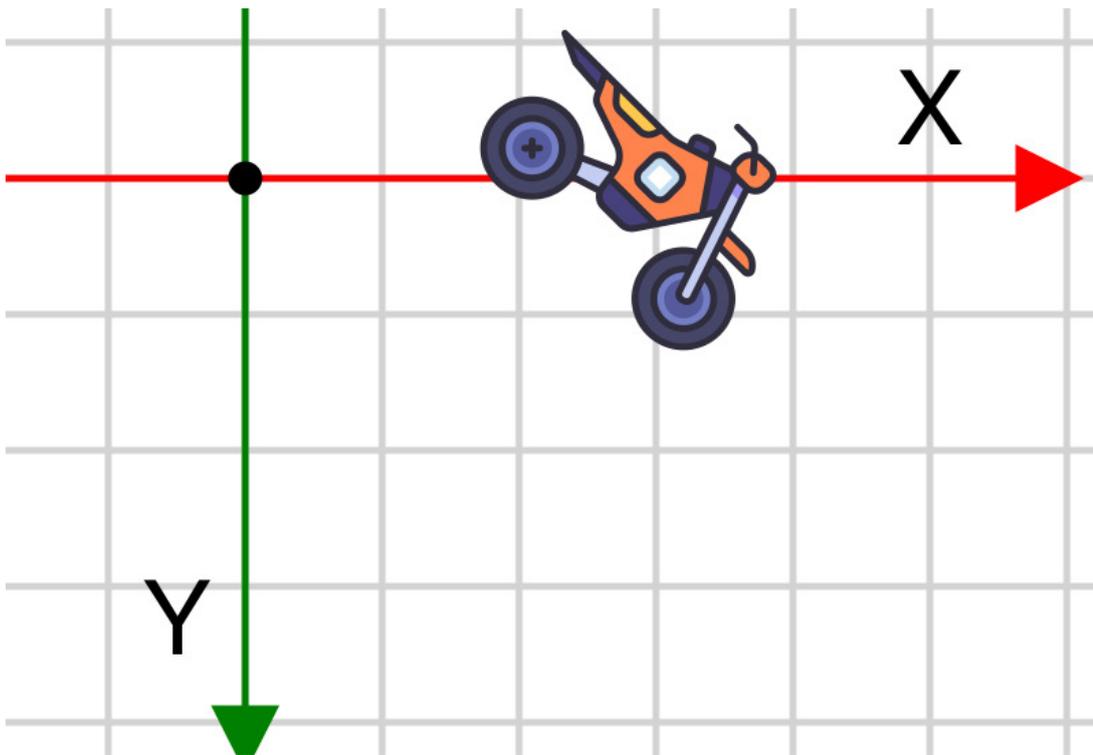
`translate(3 0)`



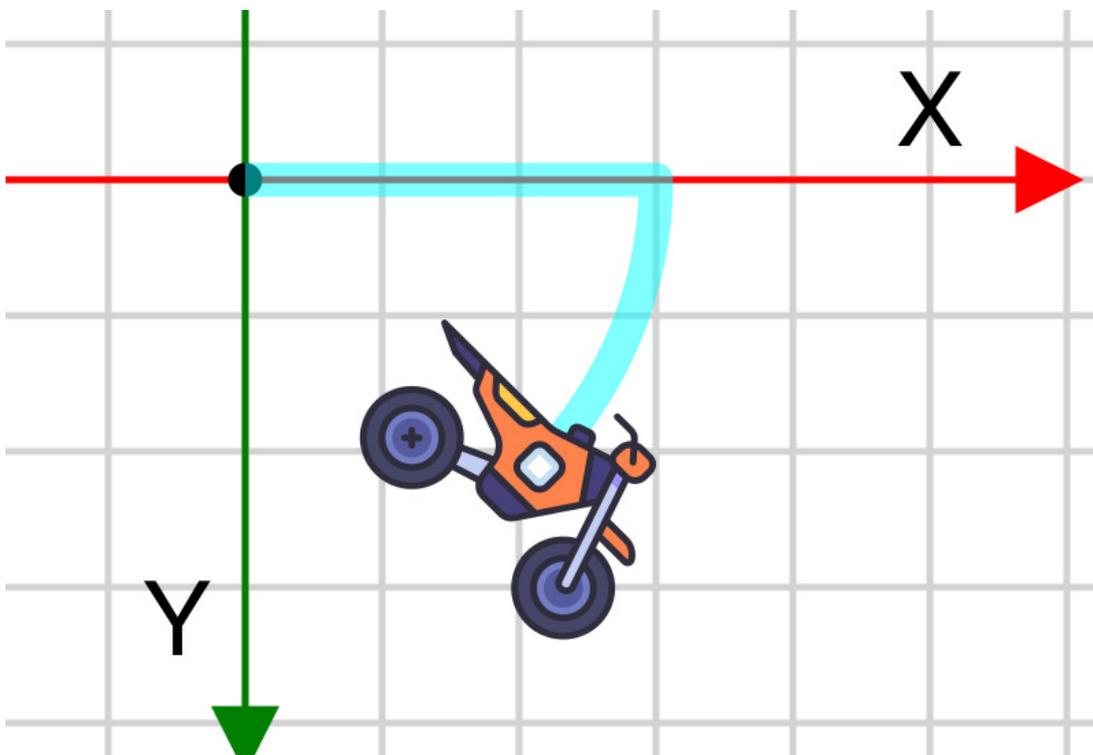
1.2.20. Ordre des transformations

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 $(0,0)$.

1.2.21. 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.

1.2.22. 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.

1.3. Programmation JavaScript

1.3.1. 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.

1.3.2. 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>
```

1.3.3. 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
}
```

1.3.4. 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)
}
```

1.3.5. Suppression d'un élément

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

1.3.6. 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.

1.3.7. 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)
```

1.4. Bibliothèque GSAP

1.4.1. Présentation

GSAP, [GreenSock Animation Platform](#) peut être employée dans tout document HTML5 pour tout animer : attributs HTML et SVG, propriétés CSS, 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 fabriqués* qui retournent un objet de type *tween* représentant une animation, c'est à dire une interpolation entre un état initial/actuel et un état final.

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

1.4.2. Remarque importante : évolutions de GSAP

GSAP évolue régulièrement. La version actuelle est la 3, voir [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 GSAP 3.

1.4.3. Un exemple complet

Dessiner un disque animé (balle.htm) :



```
<html>
<head>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/gsap/3.12.5/gsap.min.js"></script>
</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>
```

1.4.4. Analyse de l'exemple

Dans le document HTML précédent, il y a deux choses :

- Une image SVG (le *namespace* a été omis) contenant :
 - une ligne horizontale qui ne sera pas animée
 - un cercle positionné en `cx=50`, `cy=75` (en bas). C'est cette coordonnée `cy` qui va être animée.
- Un script JS réduit à une instruction :

```
gsap.to("#balle", { // identifiant de l'élément à animer
  cy: 25,           // attribut et valeur finale
  duration: 0.5,    // durée du mouvement
```

```
yoyo: true,      // faire un aller et retour
repeat: -1      // répéter à l'infini
})
```

1.4.5. Concepts de GSAP

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 JS `{attr: valeur...}` :
 - les nouvelles valeurs d'attributs, celles à atteindre à la fin de l'animation
 - le paramétrage de l'animation (voir transparents suivants).

1.4.6. Paramètres importants

Petite sélection de ce qui existe, voir [la doc](#) :

- `duration`: D durée en nombre de secondes ($D=\text{réel}>0$)
- `delay`: D nombre de secondes avant de commencer ($D=\text{réel}\geq 0$)
- `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 soit `-1`, soit supérieur à 0,

```
gsap.to("#balle", { // identifiant de l'élément à animer
  cy: 25,           // attribut et valeur finale
  duration: 0.5,    // durée du mouvement
  yoyo: true,       // faire un aller et retour
  repeat: -1        // répéter à l'infini
})
```

- `ease`: au format *"fonction.bords"*, p. ex. `"sine.inOut"` spécifie la dérivée du mouvement : accélérations et freinages.
 - *fonction* : il y a de nombreuses possibilités, 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.
 - *bords* : ce mot-clé indique quelles limites sont concernées
 - * `in` : la fonction sera appliquée seulement au début de l'animation
 - * `inOut` : la même fonction pour le début et la fin de l'animation
 - * `out` : fonction appliquée à la fin de l'animation

1.4.7. 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, ...  
})
```

1.4.8. Remarques sur l'animation

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

C'est à dire que dans l'exemple précédent, au lieu de modifier `cy="75"`, GSAP ajoute un attribut `style="cy: Npx;"`. 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(s) à animer directement  
  ...  
})
```

1.4.9. 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 car il y a mieux.

1.4.10. 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.

1.4.11. 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 supplémentaire, 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 pour spécifier la position des animations les unes par rapport aux autres.

1.4.12. Imbrication des chronologies

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
TL.add(TL2, "-=1") // démarre 1 sec avant la fin de TL1
```

1.4.13. Méthodes de GSAP

GSAP propose ces trois méthodes :

- `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

Tout est dans leur paramétrage.

1.4.14. 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) {  
    ... actions en fonction de time = temps écoulé  
}  
  
gsap.ticker.add(display)
```

La fonction `display` peut, p. ex., modifier un ensemble complexe d'attributs en fonction du temps écoulé, comme définir une position en fonction d'une équation de mouvement.

NB: `gsap.to` ne permet que d'interpoler des attributs entre une valeur initiale et une valeur finale, tandis que ce mécanisme permet de créer/supprimer des éléments, faire des calculs, etc.

1.4.15. `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, mais pas régulière.

1.4.16. Extensions de GSAP

GSAP propose de nombreuses [extensions](#), certaines gratuites, d'autres payantes (adhésion au *club*), pour des animations complexes, voir la liste sur [cette page](#).

- gestion du défilement : [ScrollTrigger](#), [ScrollTo](#)

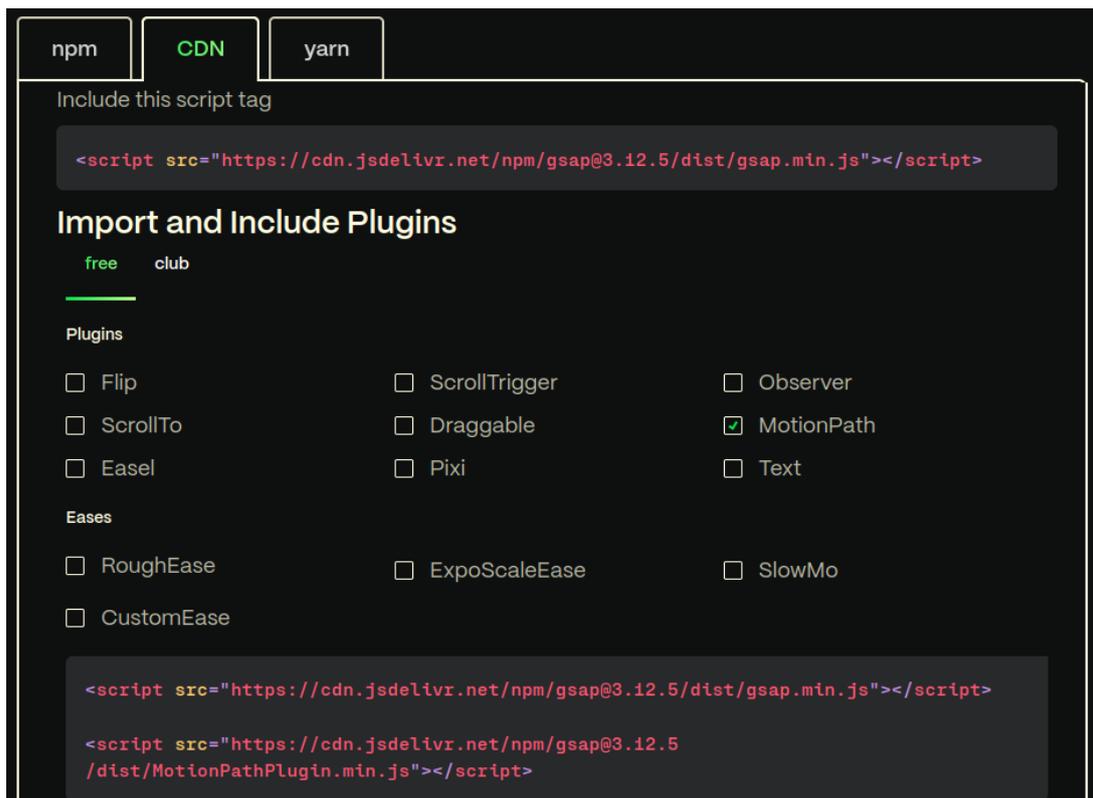


Figure 6: formulaire de choix d'installation

- physique 2D : [Physics2D](#), [InertiaPlugin](#)
- suivi de chemins : [MotionPath](#), [BezierPlugin](#)
- transformation de graphiques : [MorphSVG](#) (payant), [DrawSVG](#)
- animation CSS : [CSSRulePlugin](#), [CSSPlugin](#)

1.4.17. Installation des extensions

Utiliser [ce formulaire](#) pour connaître les inclusions CDN pour les scripts nécessaires.

1.4.18. Exemple : suivi de chemins avec le plugin MotionPath

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.

1.4.19. 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,
})
```

1.4.20. 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

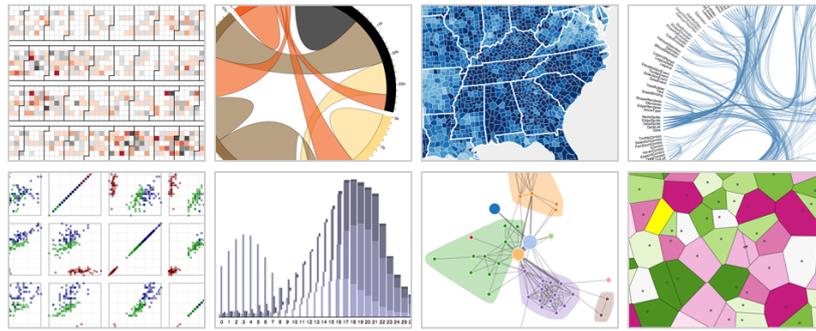


Figure 7: Exemples de visualisations

Semaine 2

d3.js

d3.js = *Data Driven Documents JavaScript library*

C'est une bibliothèque JavaScript ES6 permettant de créer des diagrammes de haute qualité et interactifs.

Elle est destinée aux scientifiques voulant visualiser des informations numériques.

Mais pas seulement, car cette API est accessible à une autre profession, intermédiaire entre science, journalisme et informatique, le *data journalisme*. C'est un métier qui consiste à expliquer le plus simplement possible, avec des graphiques réactifs, des informations numériques complexes.

Voir par exemple, [cet entretien](#).

2.1. Introduction

2.1.1. Concepts généraux

d3.js est une API de bas niveau, créée en 2011 par Mike Bostock dans le cadre de sa thèse sur la visualisation de données. Il a proposé des outils de manipulation du DOM (HTML ou image SVG) pour créer et animer des graphiques.

- d3.js : permet de tout faire, mais avec beaucoup de travail
- autres : tout est déjà fait, sauf ce dont on a besoin.

d3.js est destiné à faire des visualisations innovantes, impossibles à faire avec des API ordinaires.

2.1.2. Exemples à aller voir

Quelques exemples dans le [New York Times](#) :

- [corporate-taxes](#)
- [global business asia](#)
- [congrès républicain](#)

Quelques exemples avec les sources : [galerie observablehq](#)

Voir aussi les remarquables pages perso de Mike Bostock <https://bost.ocks.org/mike/> et <https://observablehq.com/@mbostock>. Par exemple, l'illustration de cet [algo de mélange](#).

2.1.3. Concepts de d3.js

On dispose de données : liste de n-uplets, tableau, arbre, etc.

Chaque information est dessinée avec des éléments SVG, ex: barres dans un histogramme, points ou chemins dans un diagramme, etc.

On doit programmer la génération de ces éléments. Pour cela, on met en place une association très forte entre les données et leur représentation. Tout changement dans les données entraîne un changement graphique.

Il y a de nombreux modules pour ajouter des graduations, des légendes et aussi gérer les interactions avec l'utilisateur.

2.1.4. Points forts, points faibles

- d3.js optimise les changements provoqués par des données dynamiques. Tout n'est pas redessiné à chaque changement dans les données.
- d3.js demande du travail pour faire un graphique quel qu'il soit. Il faut bien connaître la norme SVG et bien comprendre JavaScript (lambdas et fermetures). Pour des graphiques standard, il existe d'autres bibliothèques, plus faciles. Certaines sont même basées sur d3.js. Cette dernière est appropriée quand rien n'existe pour le graphique voulu.
- d3.js évolue vite (v7) et de nombreux tutoriels sont devenus des antiquités (v5 et antérieures).

2.2. Mécanismes de d3.js

2.2.1. Installation dans une page HTML

Avec un CDN (*content delivery network* ou réseau de diffusion de contenu) :



```
<!DOCTYPE html>
<html>
<head>
  <!-- d3.js -->
  <script src="https://cdn.jsdelivr.net/npm/d3@7"></script>
</head>
<body>
  ...
```

Les scripts de dessin seront plutôt placés dans le `<body>` ou alors déclenchés par un événement `onload`.

2.2.2. Sélection d'éléments

C'est la fonctionnalité de base de d3.js. Son but est de réunir certains éléments du DOM pour les modifier.

Une *sélection* dans d3.js est une liste d'éléments (balises du DOM, HTML ou SVG). Cette liste peut être vide.

Deux méthodes de sélection à connaître :

- `d3.select(sélecteur)` retourne une sélection contenant le premier élément qui correspond au sélecteur.
- `d3.selectAll(sélecteur)` retourne une sélection contenant tous les éléments qui correspondent au sélecteur, dans l'ordre du document.

```
const svg = d3.select("#dessin")
const textes = d3.selectAll("text")
```

Les sélections peuvent être enchaînées, car toute sélection possède également ces deux méthodes, `select()` et `selectAll()`. Le résultat est l'ensemble de tous les descendants désignés.

```
d3.select("#dessin")
  .selectAll("g")
  .selectAll("rect")
```

sélectionne tous les éléments `<rect>`, enfants de tous les `<g>` qui sont dans l'élément identifié par `#dessin`.

2.2.3. Parent d'une sélection

Chaque sélection est associée à l'élément parent dont elle est tirée. Le parent d'une sélection est l'élément du DOM qui contient cette sélection.

Dans la sélection d'éléments `<rect>` suivante, le parent de la sélection sont les éléments `<g>`.

```
d3.select("#dessin")
  .selectAll("g")
  .selectAll("rect")
```

Ce parent est utilisé par exemple pour ajouter de nouveaux éléments, supprimer certains éléments de la sélection, etc. Et, par exemple, quand la sélection est vide, on peut tout de même ajouter des éléments, car d3.js se souvient de l'élément parent.

2.2.4. Modifications des éléments sélectionnés

On peut modifier les attributs, la classe et le style par des méthodes spécifiques :

- `sélection.attr(nom, valeur)` crée ou modifie l'attribut
- `sélection.classed(nom(s), true ou false)` ajoute ou retire la ou les classes indiquées
- `sélection.style(nom, valeur)` crée ou modifie le style

Si *valeur* est null, alors l'attribut ou style est enlevé.

```
const svg = d3.select("#dessin")
svg.selectAll("text") // agit sur tous les <text> du dessin
  .attr("x", 10)
  .classed("titre important", true) // ajoute ces classes
  .style("fill", "Gold")
  .style("stroke", null) // supprime le style stroke
```

On peut aussi modifier le contenu des éléments d'une sélection :

- `sélection.text(valeur)` crée ou modifie le contenu texte de la sélection. Si *valeur* est null, alors le texte est enlevé.
- `sélection.append(nom)` ajoute un sous-élément `<nom>` et retourne cet élément en tant que nouvelle sélection
- `sélection.remove()` supprime les éléments de la sélection

```
svg.select("text#reponse")
  .text("C'est pas faux.")
const g = svg
  .append("g") // nouvelle sélection
  .attr("transform", "rotate(45)")
d3.select("#alerte")
  .remove()
```

2.2.5. Convention d'indentation

Il est recommandé d'indenter :

- 2 espaces pour une méthode qui retourne une nouvelle sélection
- 4 espaces pour une méthode qui retourne la sélection courante

```
d3.select("body") // sélection = l'élément <body>
  .append("svg") // <svg> ==> nouvelle sélection
  .attr("width", 960)
  .attr("height", 360)
  .append("g") // crée un <g> dans le <svg>
  .attr("transform", "translate(20,20)")
  .append("rect")
  .attr("width", 920)
  .attr("height", 360)
```

NB: il serait plus lisible de séparer ces instructions en plusieurs parties.

2.2.6. Valeurs fournies aux méthodes

Les méthodes `attr()`, `style()`, `text()` demandent un nom et une *valeur*. Elle peut être une constante, chaîne ou nombre et alors c'est cette valeur qui est affectée à tous les éléments de la sélection.

Ça peut aussi être une *fonction*. Elle est alors appelée pour chaque élément de la sélection, et peut donc retourner des valeurs distinctes. Elle reçoit deux paramètres : la donnée courante et l'indice de l'élément dans la sélection.

```
svg.selectAll("text")           // sélection des <text>
  .data([2, 3, 5, 7, 11, 13, 17]) // voir plus loin
  .text((d, i) => `v${i} = ${d}`) // contenu des <text>
```

Si c'est une vraie fonction, pas une lambda, alors `this` est lié à l'élément courant du DOM.

2.2.7. Création/modification/suppression automatique d'éléments

Dans l'exemple précédent, il faut qu'il y ait préalablement exactement autant d'éléments `<text>` que de valeurs fournies à `data()`.

La méthode `join(élément)` permet d'adapter les éléments du DOM aux données. Elle crée des éléments s'il en manque, et supprime ceux qui sont en trop.

```
svg.selectAll("text")
  .data([2, 3, 5, 7, 11, 13, 17])
  .join("text")           // crée/supprime des <text>
  .text((data, i) => `valeur ${i} = ${data}`)
```

Les méthodes `selectAll()`, `data()` et `join()` travaillent ensemble.

2.2.8. Méthode `data()` approfondie

Reprenons le début de ce code dans un contexte complet :

```
<svg id="dessin" viewBox="..."></svg> // vide

<script>
const svg = d3.select("#dessin") // élément <svg>
svg.selectAll("text")           // sous-éléments <text>
  .data([2, 3, 5, 7, 11, 13, 17]) // données à associer
```

C'est assez bizarre au premier regard. On sélectionne tous les éléments `<text>` de l'élément `<svg>`, or il n'y en a aucun.

Oui, il n'y en a aucun, mais la sélection mémorise l'élément parent de ces `<text>`, c'est `<svg>`, donc la méthode `data(valeurs)` sait qu'il va falloir y créer 7 éléments enfants, un pour chaque donnée.

NB: à ce stade, ces nouveaux éléments sont indéfinis.

`sélection.data(valeurs)` retourne une nouvelle sélection, celle des éléments existants de la sélection qui ont été liés avec les valeurs fournies. Cette sélection d'éléments existants est appelée *update*.

Elle retourne aussi deux autres sélections appelées *enter* et *exit*, mais elles ne sont pas directement visibles.

- *enter* est la sélection des nouveaux éléments (entrants), encore fictifs (*placeholders*) qui seront rajoutés aux existants pour correspondre aux données supplémentaires,

- *exit* est la sélection des éléments existants qui doivent être supprimés pour correspondre aux données.

En fait, les trois sélections voyagent ensemble, mais les méthodes `attr()`, `style()`, `text()`, etc. s'adressent seulement à *update*.

2.2.9. Sélections *enter* et *exit*

enter et *exit* sont des sortes de canaux cachés dans une sélection. On peut y accéder avec les méthodes `enter()` et `exit()` :

```
svg.selectAll("text")
  .data([2, 3, 5, 7, 11, 13, 17, 19, 23, 29])
  .enter()           // la sélection courante devient «enter»
  .append("text")   // ajoute un élément <text>
  .attr("fill", "green")
  .text((d, i) => `valeur ${d}`)
```

```
svg.selectAll("text")
  .data([11, 13, 17])
  .exit()           // la sélection courante devient «exit»
  .remove()        // supprime ces éléments
```

Les anciennes versions de d3.js fonctionnaient comme ça.

En fait, si on est sûr de uniquement créer des éléments, on peut n'utiliser que la sélection *enter*. Inversement, si on est sûr de ne faire qu'en supprimer, on peut utiliser seulement *exit* :

```
let points = [{"x": 1, "y": 4}, {"x": 7, "y": 3}, ...]
svg.selectAll("circle")
  .data(points)
  .enter()
  .append("circle")
  .attr("cx", d => d.x)
  .attr("cy", d => d.y)
  .attr("r", 2.5)
```

`cx` et `cy` sont définis par des lambdas ; leur paramètre `d` est l'un des objets de `points` ; ces objets ont deux propriétés, `x` et `y`. C'est ainsi que l'attribut `cx` est calculé par la fonction qui à `d` retourne `d.x`.

2.2.10. Liaison des données aux éléments du DOM

En résumé, on sélectionne une partie du DOM par exemple des éléments `<text>` ou `<rect>` dans un SVG :

```
svg.selectAll("TYPE")
```

On les associe un par un à des données :

```
.data(liste de valeurs)
```

On obtient une sorte de superposition de 3 listes : les éléments à modifier *update*, ceux à ajouter *enter* et ceux à supprimer *exit*.

```
.join("TYPE") // le même TYPE qu'au selectAll
```

La méthode `join()` permet de spécifier quoi faire avec ces trois listes : créer, mettre à jour et supprimer.

2.2.11. Méthode `join()` approfondie

La méthode `sélection.join(élément)` gère les trois listes de la sélection comme on peut s'y attendre : elle supprime ce qu'il y a dans *exit*, crée des éléments pour *enter* et met à jour ceux de *update* avec ce qui suit l'appel à la méthode.

```
svg.selectAll("text")
  .data([2, 3, 5, 7, 11, 13, 17])
  .join("text") // crée/édite/supprime des <text>
  .text((data, i) => `valeur ${i} = ${data}`)
```

Après exécution de `join()`, les listes *enter* et *update* sont fusionnées et *exit* est enlevée de la sélection. Il ne reste plus que les anciens éléments conservés avec les nouveaux. On peut donc chaîner d'autres méthodes pour les modifier tous de la même manière.

Au lieu d'un nom d'élément, on peut fournir à `join()` trois fonctions (ou lambdas) *enter*, *update* et *exit* dans cet ordre.

```
svg.selectAll("text")
  .data([2, 3, 5, 7, 11, 13, 17])
  .join(
    enter => enter.append("text"),
    update => update.style("background-color", "Bisque"),
    exit => exit.remove()
  )
  .text((data, i) => `valeur ${i} = ${data}`)
```

La première lambda gère les éléments à ajouter au DOM. La deuxième s'occupe des éléments existants, et ici on les colore en orange clair. La troisième lambda supprime les `<text>` devenus inutiles. Ensuite, on affecte le contenu texte des éléments restants.

2.3. Création d'un graphique avec d3.js

2.3.1. Présentation

On veut dessiner un graphique de type barres ultra simple.

Il représente ces valeurs : 4, 8, 15, 16, 42, 7, 23.

NB: c'est l'adaptation d'une série de tutoriels de Mike Bostock, [Let's Make a Bar Chart](#)

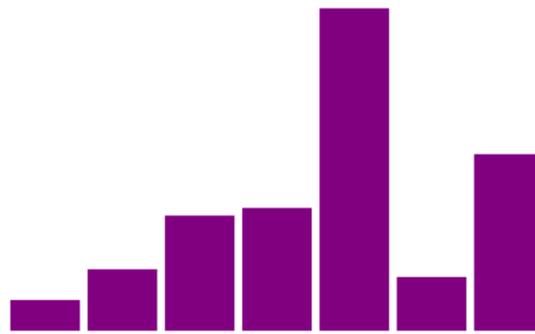


Figure 8: graphique voulu

2.3.2. Cadre général

On part de données disponibles dans une liste. Ces données doivent être liées à des rectangles SVG plus ou moins grands. 

```
<!DOCTYPE html>
<html>
<head>
  <script src="https://cdn.jsdelivr.net/npm/d3@7"></script>
</head>
<body>
  <svg id="dessin" viewBox="0 0 100 50" fill="purple"></svg>
  <script>
const valeurs = [4, 8, 15, 16, 42, 7, 23]

  </script>
</body>
</html>
```

2.3.3. Objectif

On doit obtenir quelque chose comme ça :

```
<svg id="dessin" viewBox="0 0 100 50" fill="purple">
  <rect x="0" y="46" width="9" height="4"></rect>
  <rect x="10" y="42" width="9" height="8"></rect>
  <rect x="20" y="35" width="9" height="15"></rect>
  <rect x="30" y="34" width="9" height="16"></rect>
  ...
</svg>
```

- Les rectangles ont tous la même largeur et leur position *x* va de 10 en 10.
- On reconnaît les valeurs dans les attributs *height* et aussi $(50 - \text{valeur})$ dans les *y*.

NB: les coordonnées et tailles sont relatives à la *viewBox*.

2.3.4. Réalisation

1. Obtenir une sélection des rectangles concernés.
NB: initialement, cette sélection est vide.

```
const svg = d3.select("#dessin")
svg.selectAll("rect")
```

2. Lier les données à la sélection

```
.data(valeurs)
.join("rect")
```

3. Définir les coordonnées, tailles et couleur

```
.attr("x",      (d, i) => i*10)    // d = valeurs[i]
.attr("y",      (d, i) => 50-d)
.attr("width",  9)
.attr("height", (d, i) => d)
```

2.3.5. Résultat

Voici le cœur du fichier HTML : [barchart.htm](#)

```
<body>
  <svg id="dessin" viewBox="0 0 100 50" fill="purple"></svg>
  <script>
const valeurs = [4, 8, 15, 16, 42, 7, 23]
const svg = d3.select("#dessin")
svg.selectAll("rect")
  .data(valeurs)
  .join("rect")
    .attr("x",      (d, i) => i*10)    // d = valeurs[i]
    .attr("y",      (d, i) => 50-d)
    .attr("width",  9)
    .attr("height", (d, i) => d)
  </script>
</body>
```

2.4. Compléments graphiques

2.4.1. Introduction

Le graphique précédent, simpliste, permet de comprendre les concepts de base : sélection et liaison avec des données.

- Il n'y a aucune adaptation des coordonnées à des données en plus grand nombre ou ayant une amplitude très variable. Il faudrait étirer horizontalement et verticalement en fonction des données.

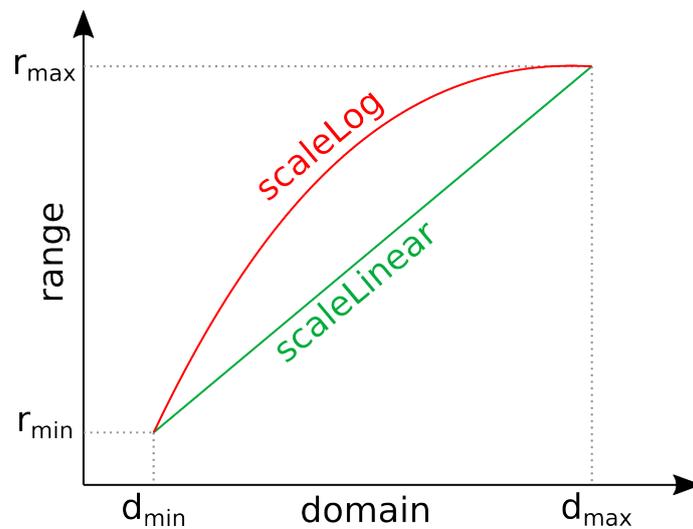


Figure 9: domain vers range

- Le graphique est très pauvre en information. Il faudrait afficher une échelle sur les deux axes. On peut aussi ajouter un titre.

Ces améliorations se font à l'aide de modules de d3.js. Ils sont assez nombreux, certains généralistes, d'autres très spécifiques.

2.4.2. Mise à l'échelle

Dans la plupart des cas, il faut adapter les dimensions du graphique à la zone disponible sur l'écran. Cela passe par des transformations de coordonnées, en général un couple translation + homothétie.

Il y a trois concepts à comprendre :

- la plage de variation des données d'entrée appelée *domain*,
- l'étendue de sortie appelée *range*, c'est ce qu'on veut comme amplitude à l'écran, par rapport à la *viewBox*
- la fonction de transfert, elle peut être linéaire (affine), logarithmique, temporelle...

Cette fonction associe une valeur du *range* à toute donnée du *domain*. Elle permet de dessiner les données dans la bonne zone de l'image.

Une échelle met en correspondance un domaine et une étendue (*range*). Le domaine est celui des données d'entrée ; l'étendue est celle des coordonnées écran. Le but est que la plus petite donnée soit affichée à la plus petite coordonnée (à gauche ou en bas), et la plus grande donnée soit affichée à la plus grande coordonnée.

Le module [d3-scale.js](#) offre de nombreux types d'échelles sous forme de méthodes de la classe `d3`, par exemple des échelles linéaires pour les axes *x* et *y* se définissent ainsi : 

```
const fX = d3.scaleLinear()  
  .domain([0, valeurs.length]) // domaine : 0..nb données  
  .range([0, 100])             // 100 = viewBox.width  
const fY = d3.scaleLinear()  
  .domain(d3.extent(valeurs)) //min(valeurs)..max(valeurs)  
  .range([0, 50])             // 50 = viewBox.height
```



Figure 10: graphique voulu

`fX` et `fY` sont à la fois des fonctions et des objets possédant des méthodes. Par exemple, `fX.range()` retourne `[0,100]` et `fY.domain()` retourne `[0, 42]`. Voir [la doc](#). Voir aussi [d3-array](#) pour `extent`, `min`, `max`...

Comme on a des données un peu particulières, une liste de valeurs, on peut employer une échelle horizontale appropriée :

```
const fX = d3.scaleBand()  
  .domain(d3.range(valeurs.length)) // [0,1,2,...,5,6]  
  .range([0, 100]) // 100 = viewBox.width  
  .padding(0.1) // espacement des bandes
```

`scaleBand` crée une échelle entre des valeurs discrètes (une liste) et un *range* continu (un axe). Ici, on veut lier les indices dans le tableau des valeurs à l'axe *X*.

La méthode `d3.range(N)` fonctionne exactement comme en Python. Elle retourne le tableau `[0..N - 1]` ([doc](#)), voir [d3-range](#).

Ces fonctions `fX()` et `fY()`, obtenues en créant des échelles, servent à calculer les coordonnées des données dans l'image :

```
.attr("x", (d, i) => fX(i))  
.attr("y", (d, i) => 50-fY(d))  
.attr("width", fX(1))  
.attr("height", (d, i) => fY(d))
```

Le graphique sera parfaitement cadré dans la `viewBox` quelles que soient les données.

NB: il est important de transformer les dimensions de la `viewBox` en constantes et les employer partout. Ce n'est pas le cas ici, avec les valeurs 50 et 100 qui se promènent un peu partout.

2.4.3. Axes et graduations

Comment rajouter des axes et des graduations ?

`d3.js` propose une *convention pour les marges* [margin convention](#), en orange ci-dessous. C'est une norme pour dessiner un graphique avec ses axes, sa légende et son titre.

Le graphique est dessiné dans le rectangle vert intérieur par une modification astucieuse des fonctions d'échelle `fX` et `fY`.

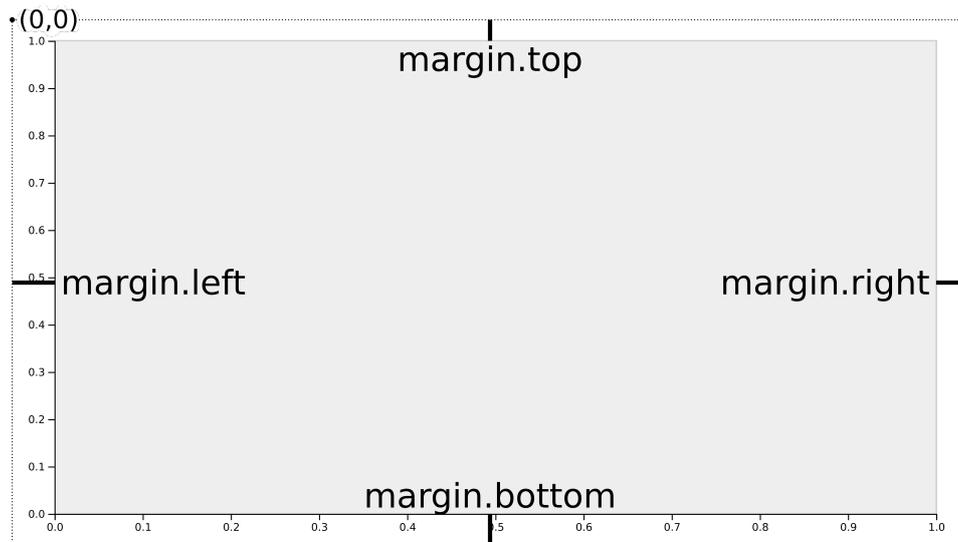


Figure 11: graphique voulu

2.4.4. Convention des marges

On commence par définir un objet qui contient les 4 marges :

```
const width = 200
const height = 100
const margin = {top: 20, right: 10, bottom: 30, left: 40}
```

Ensuite, on définit les fonctions de mise à l'échelle un peu différemment. Le *range* tient compte de la taille réduite de la zone disponible pour dessiner :

```
const fX = d3.scaleLinear() // ou d3.scaleBand()
  .domain(..) // INCHANGÉ
  .range([margin.left, width - margin.right])

const fY = d3.scaleLinear()
  .domain(..) // INCHANGÉ
  .range([height - margin.bottom, margin.top]) // ordre !!!
```

NB: pour *fY*, le *range* est inversé, car *y* va vers le bas dans un SVG.

Avec cette convention, on aura toujours ces *range*-là, quel que soit le graphique. Seuls les domaines et types d'échelles seront spécifiques.

On ne change presque rien à la manière de dessiner :

```
svg.selectAll("rect")
  .data(valeurs)
  .join("rect")
  .attr("x", (d, i) => fX(i))
  .attr("y", (d, i) => fY(d))
  .attr("width", fX(1) - fX(0))
  .attr("height", (d, i) => fY(0) - fY(d))
```

Remarquez les astuces de calcul pour calculer `width` et `height` avec les fonctions de mise à l'échelle. `fX(0)` et `fY(0)` donnent la référence du repère de dessin.

NB: les tutos de Mike Bostock regorgent d'astuces remarquables.

2.4.5. Ajout d'axes

d3.js offre des mécanismes pour ajouter des axes et des graduations à un graphique. Un axe est quelque chose d'assez complexe. Ça comprend une ligne horizontale ou verticale couvrant tout le domaine, des graduations (*ticks*) et des valeurs espacées régulièrement, le tout placé dans un groupe `<g>`.

Tout cela est créé par ces instructions :



```
svg.append("g")
  .attr("transform", `translate(0,${height-margin.bottom})`)
  .call(d3.axisBottom(fX)) // crée l'axe X en bas

svg.append("g")
  .attr("transform", `translate(${margin.left},0)`)
  .call(d3.axisLeft(fY)) // crée l'axe Y à gauche
```

2.4.6. Méthode call

La méthode `call` d'une sélection permet de simplifier l'écriture.

```
sélection.call(fonction, paramètres...)
```

est équivalente à

```
fonction(sélection, paramètres...)
```

On utilise `call` quand la sélection est complexe, et d'autre part, `call` retourne la sélection, ce qui permet de chaîner d'autres méthodes.

Ainsi :

```
svg.append("g")
  .attr("transform", `translate(0,${height - margin.bottom})`)
  .call(d3.axisBottom(fX))
```

est équivalente à :

```
d3.axisBottom(fX)(
  svg.append("g")
    .attr("transform", `translate(0,${height - margin.bottom})`)
)
```

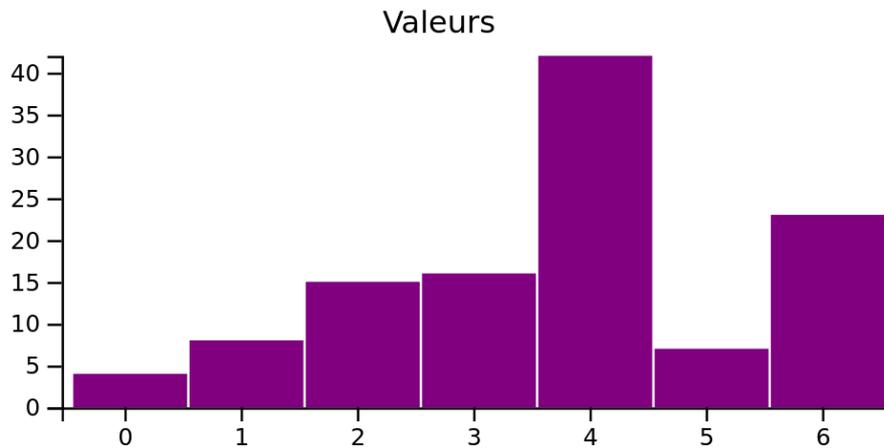


Figure 12: graphique final

parce que `d3.axisBottom(fX)` est une fonction qui crée une multitude d'éléments SVG dans la sélection (le groupe `<g>`).

Mike Bostock ajoute les axes d'une autre manière :



```
const axeX = g =>
  g.attr("transform", `translate(0,${height - margin.bottom})`)
    .call(d3.axisBottom(fX))

svg.append("g")
  .call(axeX)
```

L'axe X est défini dans une variable, `axeX`, sous forme d'une lambda et il est rajouté au graphique ultérieurement. Cela permet de rendre le dessin modulaire. On définit des lambdas pour créer les échelles, créer les axes, compléter le dessin.

2.4.7. Ajout d'un titre

Pour ajouter un titre centré en haut :



```
svg.append("text")
  .attr("text-anchor", "middle")
  .attr("x", width/2)
  .attr("y", margin.top/2)
  .text("Valeurs")
```

2.5. Données dynamiques

2.5.1. Mise à jour des données

La modification des données ou des caractéristiques d'un graphique demande de tout redessiner, c'est à dire de relancer tout le script.

Ce n'est pas du tout catastrophique, car le DOM est déjà en place grâce à l'état précédent et d3.js minimise les modifications. La méthode `data()` calcule trois listes : les nouveaux éléments, les éléments existants réutilisables et les éléments superflus. Seuls les nouveaux éléments et ceux à modifier font l'objet de calculs. Il en est de même pour les axes.

Il y a une notion de clé (voir [doc](#)) pour accompagner les données de manière à garder un ordre entre les éléments existants et les nouveaux, mais c'est trop complexe pour ce cours, voir le TD3.

2.5.2. Obtention des données

Le module [d3-fetch](#) est très utile pour télécharger des données d'un serveur, ou lire des données dans un fichier local : 

```
d3.json("https://serveur/data.json") // ou d3.json("data.json")
  .then(valeurs => dessiner(valeurs)) // dessin quand tout reçu
```

Comme toutes les méthodes de ce module, `d3.json(URL ou fichier)` est *asynchrone*. Elle retourne une *promesse* qui permet d'attendre la réception des données pour dessiner.

On peut aussi placer le chargement dans une fonction asynchrone :

```
async function getData() {
  const valeurs = await d3.json("https://serveur/data.json")
  dessiner(valeurs)
}
getData() // appel pour dessiner les données
```

2.5.3. Transitions

[d3-transition](#) permet d'animer des changements sur une sélection. C'est à dire qu'au lieu d'appliquer les changements instantanément, ils sont appliqués sur une certaine durée.

```
d3.selectAll("text")
  .transition()
  .duration(750)
  .ease(d3.easeCubicInOut)
  .style("fill", "black")
```

`sélection.transition()` retourne une nouvelle sélection dont les changements vont être progressifs. La méthode `duration(ms)` spécifie la durée en millisecondes.

La méthode `ease` permet d'indiquer l'accélération de variation, comme avec `gsap`. Voir [la doc](#) pour la liste détaillée.

On peut enchaîner des transitions successives dans la même instruction.

Un point subtil : seuls les changements situés *après* la méthode `transition` sont concernés.

Dans l'exemple suivant, on fait passer la couleur des textes de blanc à noir, puis à nouveau blanc.

```
d3.selectAll("text")
  .style("fill", "white")
  .transition().duration(500)
    .style("fill", "black")
  .transition().duration(500)
    .style("fill", "white")
```

2.6. Interactions avec l'utilisateur

2.6.1. Écouteurs sur les éléments

La méthode `sélection.on("événement", écouteur)` déclare un écouteur sur les éléments de la sélection pour l'événement indiqué. L'événement est n'importe lequel, comme `click`, `mouseover`, etc. L'écouteur est une fonction qui reçoit deux paramètres, l'événement déclencheur et la donnée associée à l'élément concerné (venant de `data()`). Dans cette fonction, `this` est lié à l'élément du DOM.

```
function onClick(event, data) {
  const that = d3.select(this) // this en tant que sélection
  that.style("fill", "red")
}

d3.selectAll("rect").on("click", onClick)
```

2.6.2. Zoom et panoramique à la souris

Zoomer et cadrer à la souris est simple. Le module `d3-zoom` analyse les actions de l'utilisateur et retourne une transformation, c'est-à-dire une composition de translations et d'homothéties, qu'il faut appliquer aux échelles et aux axes du graphique.

On commence par configurer l'élément `<svg>` pour traiter les événements souris. Ils appelleront la fonction `onZoom` :

```
svg.call(d3.zoom().on("zoom", onZoom))
```

Cette fonction, `onZoom` reçoit la transformation comme ceci :

```
function onZoom(event) {
  // appliquer event.transform aux éléments soumis au zoom
  dessin.attr("transform", event.transform)
}
```

2.6.3. Zoom sur les axes

Quand il y a des axes, c'est un peu plus complexe. Il faut fournir la transformation des échelles aux axes. Mais elle doit être distincte en x et en y .

Cela se fait avec les deux méthodes `transform.rescaleX(fX)` pour l'axe X, et `transform.rescaleY(fY)` pour l'axe Y :

```
xAxis.call(d3.axisBottom(transform.rescaleX(fX)))  
yAxis.call(d3.axisLeft(transform.rescaleY(fY)))
```

Ces méthodes appliquent la transformation limitée à l'axe concerné sur l'échelle.

2.7. Techniques avancées

2.7.1. Une fonction est un objet

d3.js emploie énormément de « choses » qui sont des fonctions un peu particulières. En JavaScript une fonction est une *sorte d'objet*, et donc on peut lui affecter des variables membres ainsi que des méthodes.

```
function F1() {  
  console.log(F1.message)  
}  
  
F1.message = "intéressant"  
F1.affiche = () => console.log("passionnant")  
  
F1()           // affiche "intéressant"  
F1.affiche()  // affiche "passionnant"
```

2.7.2. Fonctions internes

```
function F2() {  
  function action() {  
    console.log(action.message)  
  }  
  return action  
}  
  
const f2 = F2()  
f2.message = "étonnant !"  
f2()      // appelle la fonction action() et affiche "étonnant !"
```

L'appel à la fonction F2 retourne une fonction interne, `action`, qui est mise dans la variable `f2`. Ensuite, on réaffecte une variable membre de `f2` avant d'appeler `f2`. C'est étonnant car la fonction `action` trouve la variable `message` dans elle-même, alors que cette variable n'est affectée qu'après.

2.7.3. Fermetures (*closure*)

```
function F3() {
  var message = "défaut"
  function action() {
    console.log(message)
  }
  return action
}

const f3 = F3()
f3()
```

La fonction `F3` retourne une fonction interne, `action`. Or cette dernière emploie une variable qui lui est extérieure, `message`. Dans de tels cas, JavaScript mémorise cette variable dans un environnement qui accompagne la fonction interne, une *fermeture*. Ainsi, `action` a encore accès à la variable.

2.7.4. Accesseurs et modificateurs de fermetures

On peut même définir des *getters* et *setters* pour les variables.

```
function F3() {
  var message = "défaut"
  function action() {
    console.log(message)
  }
  action.getMessage = () => message
  action.setMessage = (msg) => message = msg
  return action
}

const f3 = F3()
f3.setMessage("parfait !")
console.log(f3.getMessage())
f3()
```

2.7.5. Fermetures dans d3.js

De nombreux dispositifs de `d3.js` sont des fermetures, comme par exemple les sélections, les échelles, les axes, etc.

Ce sont à la fois des objets et des fonctions. Et ces fonctions ont des paramètres qu'on précise avec des *setters*. Chacun d'eux retourne la même fermeture, de manière à pouvoir les chaîner.

```
const fX = d3.scaleLinear().domain(...).range(...)
```

parce que `d3.scaleLinear()` retourne une fermeture et elle a des *setters* programmés un peu comme ceci (voir le [source](#)) :

```
scale.domain = (interval) => {  
  domain = interval  
  return scale  
}
```

2.7.6. Ajout de méthodes pour une classe

On peut ajouter une fonction à un objet. Elle devient une méthode, mais elle reste spécifique à cet objet.

Pour ajouter une méthode à une classe existante, il faut l'ajouter dans le *prototype*, c'est à dire à la classe elle-même. Le prototype est une sorte d'objet JSON contenant toutes les méthodes et variables de classes qui sont fournies à toutes les instances.

Modifier le prototype d'une classe permet de modifier les méthodes de cette classe, après sa définition.

2.7.7. Ajout si absent

Voici un exemple d'extension de l'API de sélection. La méthode `append()` ajoute forcément un élément. Cette nouvelle méthode ne l'ajoute que s'il est absent : 

```
d3.selection.prototype.appendIfAbsent =  
function(selector, type) {  
  const element = this.select(selector)  
  return element.empty() ? this.append(type) : element  
}
```

Voici un exemple d'emploi :

```
svg.appendIfAbsent("#title", "text")  
  .attr("id", "title")  
  .text("Voici le titre")
```