On repart des concepts des TD1 et TD2 concernant la création d'images SVG et leur animation avec GSAP.

Voici le résultat à obtenir au final, figure 1. Il s'agit d'un ruban de « leds », en forme de spirale, mais d'autres formes plus simples sont prévues pour commencer. Ici, les leds sont de simples cercles gris qui deviennent rouges quand les leds sont allumées. Côté animation, les leds s'allument l'une après l'autre, mais il peut y avoir plusieurs vagues d'allumage/extinction à la suite. On peut toutes les allumer, toutes les éteindre... On peut faire ce qu'on veut. Tout est configurable, le nombre de leds, leurs couleurs, le nombre de tours de la spirale, etc.

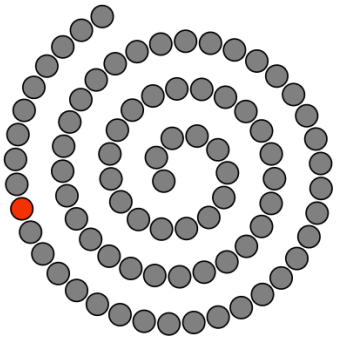

Figure 1: ruban de leds

Ce ruban n'est composé que de cercles SVG placés aux bonnes coordonnées. La complexité de l'exercice réside dans plusieurs points : le positionnement de ces cercles et leur animation.

Concernant la notation, ce TP est en nombreuses étapes : dessin d'une led, dessin de plusieurs leds, animation d'une led de différentes manières, animation du ruban, différentes formes de ruban, etc. Faites autant que possible, vous aurez des points pour chaque étape.

Attention : c'est un **travail individuel**. Demandez des explications, de l'aide, mais ne copiez en aucun cas les fichiers des collègues.

1. Préparation

Créez un nouveau dossier pour ce TP : votre compte \ Multimedia \ TP1.

On part du fichier base_gsap.htm (clic droit, enregistrer la cible du lien sous...) accompagné de svgelement.js.

Renommez `base_gsap.htm` en `tp1.htm`. Changez le contenu de la balise `<title>`, mettez TP1.

Dans l'élément `<svg>` supprimez l'attribut `height`, et modifiez `width` en `100%`, et rajoutez un attribut `viewBox="0 0 200 200"`. L'attribut `viewBox` ainsi que `width` permet de calculer `height` manquant, et il définit un système de coordonnées allant de 0 à 200 en x et y.

2. Classe `Led`, v1

Le ruban est composé de leds. Le ruban et les leds doivent être programmés sous forme de classes. Chaque instance pourra être insérée dans un élément `<svg>`.

On commence par la classe `Led`. Son constructeur doit recevoir l'élément `<svg>` en premier paramètre, nommé `parent`, et également d'autres paramètres de configuration comme la position et la taille.

Voici le début montrant le principe général :

```
/** Représente et dessine une led dans un SVG */
class Led {
  constructor(parent, x, y, radius) {
    this.led = parent.appendElement("circle", {
      // TODO coordonnées du centre
      r: radius - radius/16,
      fill: "gray",
      stroke: "black",
      'stroke-width': radius/8
    })
  }
}
```

Cette classe dessine un cercle SVG faisant exactement *radius* unités de rayon, y compris la bordure (*stroke*). Il vous reste à définir les coordonnées du centre. Relisez le TD1 ou cette documentation.

Pour tester, mettez ceci en bas de la partie `<script>` :

```
// dessin SVG
const svg = SVGelement.fromSelector("#dessin")

// dessiner une led
const led = new Led(svg, 50, 50, 16)
```

3. Classe `LedRibbon`, v1

Ensuite, on a besoin d'une classe appelée `LedRibbon` (ruban en anglais) programmée de manière à pouvoir faire ceci pour afficher un ruban de 8 leds :

```
// dessiner un ruban de 8 leds
const ribbon = new LedRibbon(svg, 8, ...)
```

Alors il y a plusieurs petits problèmes :

- on ne dit pas où créer le ruban : coordonnées de la première led,
- on ne dit pas quelle est la taille des leds,
- on ne dit pas quel est l'espacement entre les leds,
- on ne dit pas comment sont disposées les leds.

On doit donc ajouter plusieurs paramètres à ajouter au constructeur de `LedRibbon` :

```
/** Représente et dessine un ruban horizontal de leds dans un SVG */
class LedRibbon {
  constructor(parent, number, x, y, radius, spacing=0) {
    // liste des leds créées par ce ruban
    this.leds = []
    // TODO créer les leds et les mettre dans this.leds
  }
}

// dessiner un ruban de leds
const ribbon = new LedRibbon(svg, 8, 8, 8, 4)
```

Complétez la définition de la classe, sachant que les leds sont alignées horizontalement, que la première est en $(x,\ y)$ de rayon $radius$, et que la led suivante est à $(x+(radius\times 2+spacing),\ y)$, puis $(x+(radius\times 2+spacing)\times 2,\ y)$, puis $(x+(radius\times 2+spacing)\times 3,\ y)$ et qu'il y en a $number$ en tout. Commencez par comprendre les calculs à faire, ou partez carrément comme vous le sentez, du moment que les leds sont alignées horizontalement.

Il est nécessaire de mémoriser les leds dans la liste parce que c'est la classe `LedRibbon` qui construit l'animation, après la construction de l'instance, comme on le verra plus loin.

4. Classe `LedRibbon`, v2

Cette définition du constructeur présente quelques inconvénients :

- Il y a beaucoup de paramètres à fournir, sauf l'intervalle qui a une valeur par défaut.
- Ces paramètres ont le même type, on peut les confondre,
- Il faut les fournir dans l'ordre.

On se propose d'utiliser une technique JavaScript pour améliorer un peu l'usage de cette classe, des paramètres nommés avec des valeurs par défaut. C'est une syntaxe très particulière en JS. Voici un exemple. Une fonction appelée `fonctionA` demande deux paramètres obligatoires classiques, puis un objet contenant les paramètres nommés sous forme de propriétés. Les paramètres nommés qui ont une valeur par défaut sont optionnels, ceux qui n'en ont pas sont obligatoires :

```
function fonctionA(oblig1, oblig2, {
    oblig3, oblig4,
    param_opt1=val1, param_opt2=val2, param_opt3=val3
  } = {}) {
    //... utiliser oblig1, oblig2..., param_opt1, param_opt2, param_opt3 ...
    //... soit ils ont la valeur fournie à l'appel, soit la valeur par défaut ...
}

fonctionA(2, 3, {oblig3: 4, oblig4: 5, param_opt2: 6, param_opt1: 7})
```

L'appel à la fonction fournit les valeurs de tous les `oblig...` et des paramètres optionnels, dans le désordre, sauf `param_opt3`. Ce dernier aura la valeur par défaut.

Notez bien la syntaxe `{param, param=valeur, ...} = {}` dans la définition, et `{param: valeur, ...}` lors de l'appel.

Utilisez cette technique, modifiez le constructeur (ne gardez pas la première version) afin qu'on puisse créer un ruban ainsi : 

```
// dessiner un ruban de leds
const ribbon = new LedRibbon(svg, {number: 16,  x: 8, y: 8})
```

Le nombre de leds, `number` ainsi que `x` et `y` sont des paramètres obligatoires. Les valeurs par défaut des autres sont à choisir raisonnablement pour que le ruban s'affiche correctement.

5. Animation de la led, v1

On revient sur la led. On veut maintenant la faire changer de couleur à l'aide d'une animation GSAP. On voudrait pouvoir faire comme ça : 

```
// dessiner une led
const led = new Led(svg, 50, 50, 16)

// allumer la led puis l'éteindre, dans une timeline
const TLled = gsap.timeline()
led.turnOn(TLled, 1.0)           // allumage en 1 seconde
led.turnOff(TLled, 0.5, ">+2")   // extinction en 1/2 s après 2 secondes
```

C'est à dire qu'il faut ajouter deux méthodes à la classe `Led` :

- `turnOn(timeline, duration=0.1, position=">")` ajoute ce qu'il faut dans la *timeline* pour allumer la led pendant une durée *duration* (allumage progressif), et l'ajout dans la timeline se fait à la position indiquée, la chaîne `">"` signifie d'ajouter à la suite des animations précédentes.
- `turnOff(timeline, duration=0.1, position=">")` idem mais pour animer l'extinction de la led.

Programmez ces deux méthodes. Il n'y a pas besoin de mémoriser un état pour la led, seulement de changer sa couleur de `fill` entre `gray` (attention, pas `grey`) et `red` selon la méthode. Relisez bien le TD2. Voici la documentation des timeline.

Vous avez remarqué les paramètres de `timeline.to` et `gsap.to`, leur structure vous rappelle-t-elle quelque chose ?

6. Animation de la led, v2

On voudrait pouvoir animer la led *directement* avec GSAP, faire comme ceci : 

```
// dessiner une led
const led = new Led(svg, 50, 50, 16)
```

```
// allumer la led puis l'éteindre, dans une timeline
const TLled = gsap.timeline()
...

// pareil mais en manipulant la led comme si c'était un objet du DOM
TLled.to(led, {
    state: 1.0,     // 1.0 = pleinement allumée
    duration: 1.0
  })
TLled.to(led, {
    state: 0.0,     // 0.0 = totalement éteinte
    duration: 0.5
  },
  ">+2")            // voir timeline.to(target, vars, position)
```

C'est à dire qu'on veut faire avec la led comme on faisait avec la voiture du TD2, la placer en tant que *target* dans les paramètres de la méthode `to` de la *timeline*, voir la doc et définir son état comme on le fait pour des objets du DOM. Sauf que :

- La variable led ne représente pas une balise du document HTML, c'est à dire un objet du DOM, seul `led.led` en est un, l'élément `<circle>` dessiné à l'écran. Mais on veut directement utiliser `led`, pas `led.led`.
- La led n'a pas de propriété `state`, donc GSAP ne peut pas animer cette propriété. GSAP ne peut même pas appeler `led.turnOn` et `led.turnOff` car il n'y a aucun moyen de faire le lien avec ces méthodes selon la valeur d'allumage de la led.

Donc on va rajouter une *propriété* à la classe `Led` et découvrir ainsi les *setters* et *getters* en JS.

Lire ces explications. En résumé, on définit des accesseurs et modificateurs pour une classe ainsi :

```
class Truc {
  constructor(...) {
    this._champ = ...
  }

  get champ() {
    return this._champ     // et/ou n'importe quel calcul
  }

  set champ(valeur) {
    this._champ = valeur   // et/ou n'importe quelle action
}
```

Devinez pourquoi il ne faut pas donner le même nom à la variable membre qu'aux *getter* et *setter*, sinon périssez sur place. Ici, j'ai appelé `_champ` la variable membre, et `champ` les *getter* et *setter*.

Programmez ces deux méthodes spéciales pour ajouter une propriété `state` à la classe `Led` (laissez les méthodes `turnOn/Off`). C'est un nombre réel entre 0 et 1. Vous devrez interdire les valeurs hors plage, les forcer à être dans la plage 0..1.

GSAP a besoin du *getter* pour consulter l'état de départ et déterminer comment interpoler vers la valeur cible. Il sera préférable d'avoir une variable membre pour mémoriser cet état, parce qu'il sera difficile à déterminer d'après la couleur seulement.

Le gros point technique, c'est qu'il ne suffit pas, dans le *setter* de mémoriser la nouvelle valeur, mais il faut agir sur la couleur. Voici comment, si la nouvelle valeur s'appelle `value` :

```
const color = gsap.utils.interpolate("gray", "rgb(127, 255, 0)", value)
this.led.setAttribute("fill", color)
```

Comment ça, j'ai mis une autre couleur ? Remarquez que je l'ai écrite sous la forme RGB afin de permettre son interpolation avec le gris.

Si les deux types d'animations successives avec la *timeline*, par `turnOn/Off` puis changement du `state` ne marchent pas ensemble alors qu'elles fonctionnent bien indépendamment, c'est qu'il y a une bataille entre les styles et les attributs. Faites le ménage dans les méthodes `turnOn/Off` pour que ça ne modifie que l'attribut sans passer par un style.

7. Boutons pour lancer les animations

Copiez la structure des boutons du TD2 exo 2 pour avoir un bouton reset qui supprime les animations sur la led et qui l'éteint, un bouton `animation1` qui allume puis éteint avec les méthodes `turnOn/Off`, `animation2` qui anime avec uniquement la propriété `state` et un `animation3` qui fait les deux successivement. Mettez ce panneau au dessus du dessin SVG (plus pratique si l'écran est petit).

Dans la fonction `reset()`, pour supprimer toute animation, faire `gsap.killTweensOf("*")`. Et il faudrait ajouter une méthode `reset()` dans la classe `LedRibbon` pour appeler une méthode du même nom sur toutes ses leds.

Voici la méthode `reset()` de la classe `Led` à ajouter :

```
class Led {
    ...

    reset() {
        this.state = 0
        gsap.killTweensOf(this)
    }
}
```

8. Animation du ruban

On arrive au plus intéressant, animer les leds du ruban. On veut obtenir une sorte de guirlande électrique, où les leds s'allument et s'éteignent les unes après les autres en donnant l'impression d'un point qui se déplace.

Le principe est de placer tous les allumages et extinctions dans une seule timeline passée en paramètre à une méthode du ruban. On veut pouvoir faire ceci :

```
// dessiner un ruban de leds
const ribbon = new LedRibbon(svg, {number: 16, x: 8, y: 8})

// allumer puis éteindre toutes les leds les unes après les autres dans une timeline
const TLrib = gsap.timeline()
ribbon.turnOnOff(TLrib, 0.2)    // chaque led allumée 0.2 seconde
```

Il faut placer les deux dernières instructions et le commentaire dans une nouvelle fonction, `animation4()` associée à un nouveau bouton.

Puis programmez la méthode `turnOnOff(timeline, durationOn)` dans la classe `LedRibbon`. Cette méthode ajoute dans la timeline les événements d'allumage, d'extinction après une petite pause, `durationOn`, puis passe à la led suivante. Finalement, c'est un peu ce qu'il y a dans `animation2`, sauf qu'il faut le faire sur toutes les leds de la liste et qu'il y a un délai `durationOn` à prendre en compte, et qu'il faut nettement accélérer les temps d'allumage et d'extinction de chaque led (0.05s par exemple).

9. Amélioration de la classe `LedRibbon`

9.1. Critiques

Cette classe a deux responsabilités très différentes : placer les leds selon un algorithme de placement, par exemple alignées horizontalement et commander les animations dans l'ordre des leds. On souhaiterait ne garder que l'aspect gestion d'une liste de leds et animation dans la classe `LedRibbon` et envoyer ce qui est placement géométrique dans un autre dispositif.

L'une des solutions consiste à rendre la classe `LedRibbon` abstraite et à dériver des sous-classes spécialisées pour chaque forme géométrique. Il reste que les classes finales mélangent toujours les deux responsabilités : placement géométrique et gestion de la liste. Et aussi, en JS, il n'y a pas de classes abstraites, donc des risques de mauvais emploi.

On va donc utiliser une autre solution, un mécanisme JavaScript appelé *générateur*. Ce mécanisme extérieur à la classe `LedRibbon` sera responsable du placement des leds à la demande du ruban.

9.2. Générateur JavaScript

Un générateur est un objet très particulier. Ça ressemble à un itérateur car on peut lui demander plusieurs valeurs successivement et on peut faire une boucle sur ces valeurs, mais il n'y a pas de collection regroupant toutes les valeurs. Les valeurs sont calculées une par une et seulement au moment où on les demande. Voici un petit exemple, à essayer dans un coin :

```
// générateur des nombres impairs inférieurs à une limite
function* impairsInferieursA(limite) {
  let nb = 1
  while (nb < limite) {
    yield nb      // « yield » = fournir cette valeur puis s'endormir
    nb = nb + 2
```

```
  }
}
```

C'est comme une fonction qui aurait plusieurs résultats successifs. Au lieu d'écrire `return`, on met `yield` (fournir) et au lieu de `function`, on met `function*`. Chaque `yield` est comme un `return` sauf que le traitement peut reprendre ultérieurement après ce `yield`. Voici comment on utilise cette fonction spéciale :

```
// affichage de quelques nombres impairs
const generateur1 = impairsInferieursA(20)

console.log(generateur1.next().value)
console.log(generateur1.next().value)
console.log(generateur1.next().value)
```

La variable `generateur1` est affectée avec le générateur. C'est comme une collection, sauf qu'elle n'est pas créée maintenant. Le fait d'appeler `next()` fait exécuter le générateur jusqu'à un premier `yield`. On retrouve alors la valeur fournie dans le champ `value`.

Lorsqu'on rappelle la méthode `next()`, le générateur est réveillé, il reprend son travail jusqu'à son prochain `yield`. Et ainsi de suite.

La fonction `next()` appelée sur un générateur le reveille, et lui demande la valeur suivante. `next()` retourne un objet contenant deux champs, `value` avec la valeur fournie par un `yield` et `done` qui vaut `false` quand le générateur a encore au moins un autre `yield` à faire. Et `done` vaut `true` quand le générateur a fini, soit à cause d'un vrai `return`, doit quand il termine son code exécutable.

On peut facilement itérer sur les générateurs :

```
// affichage des nombres impairs inférieurs à 20
const generateur2 = impairsInferieursA(20)

for (const nombre of generateur2) {
  console.log(nombre)
}
```

Comme tout itérateur, on ne peut pas repartir au début, il faut créer un nouveau générateur. Dernier point, comme toute fonction, un générateur peut fournir n'importe quel type de valeurs, nombres, chaînes et objets. Dans le cas des objets, on fait ainsi :

```
// générateur de personnes
function* personnes() {
  yield {prenom: 'Roberto', nom: 'Rastapopoulos'}
  yield {nom: 'Tournesol', prenom: 'Tryphon', role: 'savant'}
  yield {prenom: 'Allan', nom: 'Thompson'}
}
```

```
// affichage des personnes
for (const {nom, prenom} of personnes()) {
  console.log(prenom, nom)
}
```

L'intérêt d'un générateur est de pouvoir séparer une logique qui crée des valeurs, d'une autre logique qui utilise ces valeurs. Un générateur mémorise un état d'avancement, par exemple des variables locales, pour pouvoir reprendre son travail à tout moment. En algorithmique, on appelle ça une *coroutine* (voir wikipedia). Une *routine* est une sorte de fonction ou procédure, une *sous-routine* est une sous-fonction, et une *coroutine* est une fonction qui est connectée à une autre et qui s'exécute en parallèle.

9.3. Séparation de la génération des coordonnées

Le but de cette partie du TP est de placer la génération des coordonnées des leds du ruban dans un générateur et de fournir ce générateur en paramètre du constructeur du ruban. Ainsi, le ruban s'occupe de la liste des leds, et le générateur s'occupe des calculs géométriques, et on peut programmer plusieurs générateurs pour des dispositions différentes sans rien changer dans `LedRibbon`.

Complétez ce générateur qui retourne successivement des objets `{x, y, radius}` :

```
/**
 * fournit successivement number objets {x, y, radius} donnant les coordonnées
 * {x, y, radius}, {x+interval, y, radius}, {x+2×interval, y, radius}, {x+3×interval,...
 * avec interval = radius×2 + spacing
 * @param number : nombre de coordonnées fournies au total
 * @param x,y : premières coordonnées renvoyées
 * @param spacing : intervalle entre les différents x retournés
 * @return générateur de {x, y, radius}
 */
function* HorizontalLine({number, x, y, spacing=0, radius=4} = {}) {
  // TODO boucle de number itérations {
    yield {x, y, radius}
    // TODO modifier x
  }
}
```

Il faut maintenant modifier le constructeur de `LedRibbon` pour pouvoir créer un ruban ainsi (remplacez les lignes concernées par ceci) :

```
// dessiner un ruban de leds
const ribbon = new LedRibbon(svg, HorizontalLine({
  number: 16,   // nombre de leds
  x: 8, y: 8,   // centre de la première led
}))
```

Vérifiez que l'animation 4 fonctionne toujours

Le constructeur de `LedRibbon` n'a plus que deux paramètres, `parent` et `generator`. Il ne contient plus que l'initialisation de la liste, car il appelle le générateur pour avoir les coordonnées des leds. C'est tout et c'est propre.

On propose maintenant d'expérimenter plusieurs dispositions de leds. Il suffit de programmer autant de générateurs. Leurs algorithmes sont très courants en dessin 2D.

10. Générateur pour une ligne inclinée

On veut un générateur, `PolarLine`, pour une ligne inclinée d'un certain angle, par exemple 25° comme dans la figure 2 (il reste la led du début).

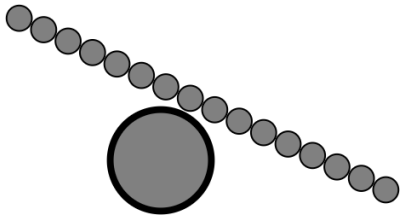

Figure 2: ligne inclinée à 25°

Voici comment on peut utiliser ce générateur :

```
// dessiner un ruban de leds alignées inclinées
const ribbon = new LedRibbon(svg, PolarLine({
  number: 16,   // nombre de leds
  x: 8, y: 8,   // centre de la première led
  angle: 25     // angle
}))
```

Comme l'autre générateur, il y a les paramètres `spacing` et `radius` avec des valeurs par défaut.

Contrairement aux apparences, ce n'est pas compliqué. À chaque tour de boucle, il faut ajouter une quantité dx à x et une quantité dy à y. Dans `HorizontalLine`, on ajoutait quelque chose qu'à x, rien à y. Ici c'est la même chose qu'on ajoute à x et y mais multipliée par le cosinus de l'angle pour x et par le sinus de l'angle pour y. Voici une esquisse de l'algorithme :

```
function* PolarLine({number, x, y, angle, spacing=0, radius=4} = {}) {
  // TODO convertir l'angle en radians
  // TODO déterminer quelle est la quantité à ajouter pour passer d'une led à l'autre
  dx = cette quantité * cos(angle)
  dy = cette quantité * sin(angle)
  // TODO boucle de number itérations {
    yield {x, y, radius}
    x += dx
    y += dy
  }
}
```

11. Générateur en cercle

On veut obtenir ceci, figure 3, (où traîne encore la première led).

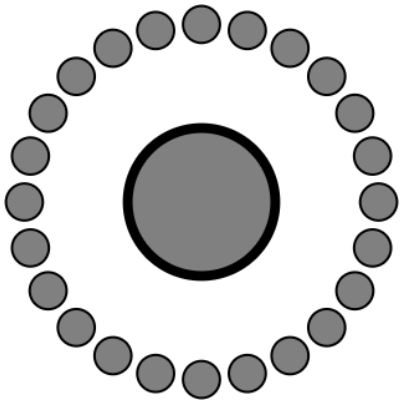

Figure 3: cercle de leds

Voici comment on peut utiliser ce générateur `Circle` :

```
const ribbon = new LedRibbon(svg, Circle({
  number: 24,      // nombre de leds
  cx: 50, cy: 50, // centre du cercle
  r: 36           // rayon du cercle
}))
```

On doit placer les leds à une distance r de $(cx,\ cy)$. C'est encore de la trigonométrie 2D. Voici une esquisse à compléter :

```
function* Circle({number, cx, cy, r, radius=4} = {}) {
  // TODO boucle de number itérations, variable i: 0..number-1
    // angle en radians
    angle = 2 pi . i / number
    x = cx + r.cos(angle)
    y = cy + r.sin(angle)
    yield {x, y, radius}
  }
}
```

À vous de faire marcher ce générateur.

12. Générateur en spirale

Pour obtenir la disposition de la figure 1, au début du sujet, il suffit de faire comme avec le cercle, mais avec plusieurs tours et en posant que r augmente linéairement au long des itérations :

```
function* SpiralV1({number, cx, cy, turns, pitch, radius=4} = {}) {
  // TODO boucle de number itérations, variable i: 0..number-1
    // angle en radians
    angle = turns . 2 pi . i / number
    // distance au centre
    r = turns . pitch . i / number
    x = cx + r.cos(angle)
    y = cy + r.sin(angle)
    yield {x, y, radius}
  }
}

// dessiner un ruban de leds
const ribbon = new LedRibbon(svg, SpiralV1({
  number: 48,       // nombre de leds
  cx: 120, cy: 50,  // centre de la spirale
  turns: 3,         // nombre de tours
  pitch: 12         // écart entre les spires
}))
```

Il y a alors un petit souci, c'est que les leds du centre sont les unes sur les autres. Le problème est que l'angle entre les leds est constant et la disposition des leds ressemble à des rayons de vélo. On doit modifier les calculs pour que l'espacement entre les leds soit constant. Voici un nouvel algorithme :

```
function* SpiralV2({number, cx, cy, pitch, spacing, radius=4} = {}) {
  angle = 0
  // TODO boucle de number itérations, variable i: 0..number-1
    // évolution de l'angle, en radians
    angle = racine carrée(2.spacing²/pitch + angle²)
    // distance au centre
    r = pitch . angle / (2 pi)
    x = cx + r.cos(angle)
    y = cy + r.sin(angle)
    yield {x, y, radius}
  }
}

// dessiner un ruban de leds
const ribbon = new LedRibbon(svg, SpiralV2({
  number: 44,       // nombre de leds
  cx: 120, cy: 50,  // centre de la spirale
  pitch: 12,        // écart entre les spires
  spacing: 7,       // écart entre les leds
}))
```

Il y a un calcul pour trouver l'angle tel que l'écart d'abscisse curviligne soit constant et égal à

`spacing`. Par contre avec ce générateur, on ne peut pas prévoir le nombre de tours et l'angle final. On doit ajouter ou enlever des leds pour que ça tombe comme on veut.

13. Remise du travail

Déposez votre fichier `tp1.htm` sur Moodle dans la zone de dépôt du TP. Vous ne devez en aucun cas copier le travail, même partiel d'un camarade. Les enseignants sont là pour vous aider à avancer du mieux possible, mais sans vous donner directement la solution.