



Guide des animations SVG (SMIL)

17 DÉCEMBRE 2014 on CSS, HTML, Design, SVG, Avancé, Transformations, Animations

Par Sara Soueidan

Introduction de Chris Coyier : Sara a le don de plonger au coeur des fonctionnalités du web et de nous les expliquer à nous autres, simples mortels. Ici, elle va creuser au plus profond de SMIL (et ses amis) et de la syntaxe d'animation de SVG pour nous donner ce guide *épique*.

📖 *Note du traducteur épuisé* : guide épique s'il en est, et qui peut paraître ardu, surtout dans ses premiers paragraphes. Mon conseil : regardez les animations (en commençant ici par exemple), lisez les paragraphes dans le désordre, puis relisez dans l'ordre.

Généralités

Les graphiques SVG peuvent être animés au moyen d'**éléments d'animation**. Les éléments d'animation ont été définis à l'origine dans la spécification d'animation SMIL (Synchronized Multimedia Integration Language). Ces éléments comprennent :

- `<animate>` - qui vous permet d'animer des attributs et propriétés scalaires sur une période de temps donnée.
- `<set>` - qui est un raccourci pratique d' `<animate>`, utile pour assigner des valeurs d'animation à des attributs et propriétés non numériques, telles que la propriété visibilité.
- `<animateMotion>` - qui déplace un élément le long d'un chemin.
- `<animateColor>` - qui modifie la valeur de couleur d'attributs ou de propriétés dans le temps. Notez que l'élément `<animateColor>` est désormais obsolète, on utilise simplement l'élément `<animate>` en ciblant les propriétés qui peuvent prendre des valeurs de couleurs. Il est toujours présent dans la spécification SVG 1.1, mais il est clairement indiqué qu'il est obsolète — et il est absent de la spécification SVG 2.

En plus des éléments d'animation définis dans la spec SMIL, SVG inclut des extensions compatibles avec ladite spécification. Ces extensions incluent les attributs étendant la fonctionnalité de l'élément `<animateMotion>` et des éléments d'animation supplémentaires. Les extensions SVG comprennent :

- `<animateTransform>` - vous permet d'animer l'un des attributs de transformation SVG dans le temps, comme l'attribut `<transform>`.
- `<path>` (*attribut*) - permet à toute fonctionnalité de la syntaxe de chemin SVG d'être spécifiée dans un attribut de chemin de l'élément `<animateMotion>` (l'animation SMIL permet seulement

un sous-ensemble de cette syntaxe à l'intérieur d'un attribut de chemin). Nous reviendrons sur `<animateMotion>` dans une section qui suit.

- `<mpath>` - utilisé en conjonction avec l'élément `<animateMotion>` pour référencer un chemin et indiquer qu'il servira de... chemin à une animation. L'élément `<mpath>` est inclus à l'intérieur de l'élément `<animateMotion>` avant la balise fermante.
- `<keypoints>` (*attribut*) - utilisé comme attribut pour `<animateMotion>` afin de fournir un contrôle précis de la vitesse des animations sur les chemins.
- `<rotate>` (*attribut*) - utilisé comme attribut pour `<animateMotion>` afin de contrôler si un objet est automatiquement pivoté de façon à ce que son axe des x pointe dans la même direction (ou la direction opposée) au vecteur tangent directionnel du chemin. Cet attribut est essentiel pour que le mouvement le long d'un chemin fonctionne comme le souhaitez. Nous verrons cela plus en détail dans la section `<animateMotion>`.

Les animations SVG peuvent être de nature similaire aux animations et transitions CSS. On crée des keyframes, les objets se déplacent, les couleurs changent etc. Cependant, elles peuvent faire certaines choses que les animations CSS ne permettent pas de réaliser, ce que nous allons voir tout à l'heure.

Pourquoi utiliser les animations SVG?

Les SVG peuvent être stylés et animés avec CSS (slides). À la base, toute animation de transformation ou de transition qui peut être appliquée à un élément HTML peut l'être à un élément SVG. Mais certaines propriétés SVG qui ne peuvent pas être animées avec CSS peuvent l'être avec SVG. Par exemple un chemin SVG vient avec un ensemble de **données** (un attribut `d=""`) qui définit la forme de ce chemin. Cette donnée peut être modifiée et animée avec SMIL, mais pas avec CSS. C'est pourquoi les éléments SVG sont décrits au travers d'un ensemble d'attributs appelés les *attributs de présentation SVG*. Certains d'entre eux peuvent être déterminés, modifiés et animés avec CSS, d'autres non.

Autrement dit, de nombreux effets et animations ne peuvent pas être obtenus avec CSS. Pour combler ces manques, on peut utiliser JavaScript ou les déclarations dérivées de SMIL.

Si vous préférez utiliser JavaScript, je recommande [snap.svg](#) de Dmitry Baranovskiy qui est décrit comme le "jQuery du SVG". Voici quelques exemples de ce qu'on peut faire.

Si vous préférez une approche plus déclarative, vous pouvez utiliser les éléments d'animation SVG, nous allons les passer en revue dans ce guide !

Un autre avantage de SMIL sur les animations JavaScript est que les animations JS ne fonctionnent pas lorsque le SVG est embarqué (*embedded*) en tant qu'`img` ou utilisé comme `background-image` dans CSS. Les animations SMIL fonctionnent, elles, dans les deux cas (ou le devraient, il faut aussi tenir compte des limitations des navigateurs). C'est un grand avantage à mon sens et vous pourriez choisir SMIL pour cette bonne raison. Cet article est conçu comme un guide pour vous aider à vous lancer dans SMIL dès maintenant.

Compatibilité navigateurs et fallbacks

La compatibilité navigateur des animations SMIL est tout à fait convenable. Elles fonctionnent dans tous les navigateurs à l'exception d'Internet Explorer et Opera Mini. Pour une étude complète de la compatibilité, vous pouvez vous référer au tableau de [Can I Use](#).

Si vous avez besoin de solutions de repli (*fallback*) pour les animations SMIL, vous pouvez tester la compatibilité navigateurs à la volée avec [Modernizr](#). Lorsque SMIL n'est pas compatible, vous pouvez fournir une solution de repli (animations JavaScript, expérience utilisateur différente, etc.).

Spécifier la cible de l'animation avec `xlink:href`

Quel que soit celui des quatre éléments d'animations que vous choisirez, il vous faut spécifier la cible de l'animation définie par cet élément.

Pour spécifier une cible, vous pouvez utiliser l'attribut `xlink:href`. L'attribut prend une référence URI de l'élément cible de l'animation. **L'élément cible doit faire partie du fragment de document SVG courant.**

```
//SVG  
<rect id="cool_shape" ... />
```

```
<animation xlink:href="#cool_shape" ... />
```

Si vous avez déjà rencontré des éléments d'animation SVG, vous les avez probablement vus imbriqués à l'intérieur de l'élément qu'ils sont supposés animer. C'est également possible selon la spécification :

Si l'attribut `xlink:href` n'est pas fourni, l'élément cible sera l'élément immédiatement parent de l'élément d'animation courant

```
//SVG  
<rect id="cool_shape" ... >  
  
  <animation ... />  
  
</rect>
```

Donc si vous voulez “encapsuler” l'animation à l'intérieur de l'élément auquel elle s'applique, vous le pouvez. Et si vous préférez spécifier les animations ailleurs dans votre document, vous pouvez également le faire, et spécifier la cible de chaque animation en utilisant `xlink:href` – les deux façons de faire fonctionnent.

Spécifier la propriété cible de l'animation avec `attributeName` et `attributeType`

Tous les éléments d'animation partagent un autre attribut : `attributeName`, qu'on utilise pour spécifier le nom de l'attribut que vous animez.

Par exemple, si vous voulez animer la position du centre d'un cercle `circle` sur l'axe des x, vous pouvez le faire en spécifiant `cx` comme valeur de l'attribut `attributeName`.

`attributeName` prend une valeur unique, il ne prend pas de liste de valeurs, par conséquent vous ne pouvez animer qu'un seul attribut à la fois. Si vous voulez animer plus d'un attribut, vous devez définir plus d'une animation pour l'élément. Sur ce point au moins, CSS a un avantage sur SMIL. Mais là encore, en raison des valeurs possibles pour les autres attributs d'animation (que nous allons voir ensuite), il est logique de ne définir qu'un seul nom d'attribut à la fois, sans quoi les autres valeurs d'attributs pourraient devenir trop complexes à gérer.

Lorsque vous spécifiez le nom d'attribut, vous pouvez ajouter un préfixe XMLNS (espace de nom XML) pour indiquer l'espace de nom de l'attribut. L'espace de nom peut également être spécifié en utilisant l'attribut `attributeType`. Par exemple, certains attributs font partie de l'espace de nom CSS (ce qui signifie que l'attribut peut être trouvé comme propriété CSS) et d'autres sont uniquement XML. Vous pouvez consulter [une table de ces attributs ici](#). Tous les attributs SVG ne figurent pas dans cette table, seulement ceux avec lesquels CSS fonctionne. Certains d'entre eux sont déjà disponibles comme propriétés CSS.

Si la valeur de `attributeType` n'est pas explicitement définie, ou si elle est définie comme `auto`, le navigateur doit d'abord chercher dans la liste des propriétés CSS un nom de propriété correspondant, et s'il n'en trouve pas, chercher le nom d'espace par défaut pour l'élément.

Par exemple, le code suivant anime l'`opacity` d'un rectangle SVG. Puisque l'attribut `opacity` existe aussi en tant que propriété CSS, l'`attributeType` est défini à partir de l'espace de nom CSS :

```
//SVG
<rect>
  <animate attributeType="CSS" attributeName="opacity"
    from="1" to="0" dur="5s" repeatCount="indefinite" />
</rect>
```

Nous allons voir les autres attributs d'animation dans les exemples qui suivent. Sauf indication contraire, tous les attributs d'animation sont communs à tous les éléments d'animation.

Animer l'attribut d'un élément

Commençons en déplaçant un cercle d'une position à une autre. Pour ce faire, nous allons modifier la valeur de son attribut `cx` qui spécifie la position de son centre sur l'axe des x.

Nous allons utiliser l'élément `animate`. Cet élément est utilisé pour animer un attribut à la fois. Les attributs peuvent prendre des valeurs numériques et les couleurs sont généralement animées avec `animate`. Pour une liste des attributs qui peuvent être animés, vous pouvez vous référer à [cette table](#).

Si l'on veut modifier une valeur sur une période de temps donnée, on utilise les attributs `from`, `to` et `dur`. Par ailleurs, si vous avez besoin de spécifier quand l'animation doit démarrer, vous utiliserez l'attribut `begin`.

```
//SVG
<circle id="my-circle" r="30" cx="50" cy="50" fill="orange" />

<animate
  xlink:href="#my-circle"
  attributeName="cx"
  from="50"
  to="450"
  dur="1s"
  begin="click"
  fill="freeze" />
```

Dans l'exemple ci-dessus, nous avons défini un cercle, puis nous appelons une animation sur ce cercle. Le centre du cercle se déplace de sa position initiale (50 unités) vers sa position finale (450 unités) sur l'axe des x.

La valeur de `begin` est définie comme `click`. Cela signifie que le cercle se déplacera lorsqu'on cliquera dessus. Vous pouvez également définir cette valeur comme une unité de temps, par exemple `begin="0s"` démarrera l'animation dès que la page est chargée. Vous pouvez **retarder l'animation** en donnant une valeur de temps positive — par exemple `begin="2s"` qui démarrera l'animation deux secondes après le chargement de la page.

Ce qui est encore plus intéressant avec `begin`, c'est que vous pouvez définir des valeurs telles que `click + 1s` pour démarrer une animation **une seconde après que l'élément ait été cliqué**. De plus, vous pouvez utiliser d'autres valeurs qui vous permettent de synchroniser les animations sans avoir à calculer la durée et les retards des autres animations. Nous verrons cela tout à l'heure.

L'attribut `dur` est similaire à son équivalent CSS `animation-duration`.

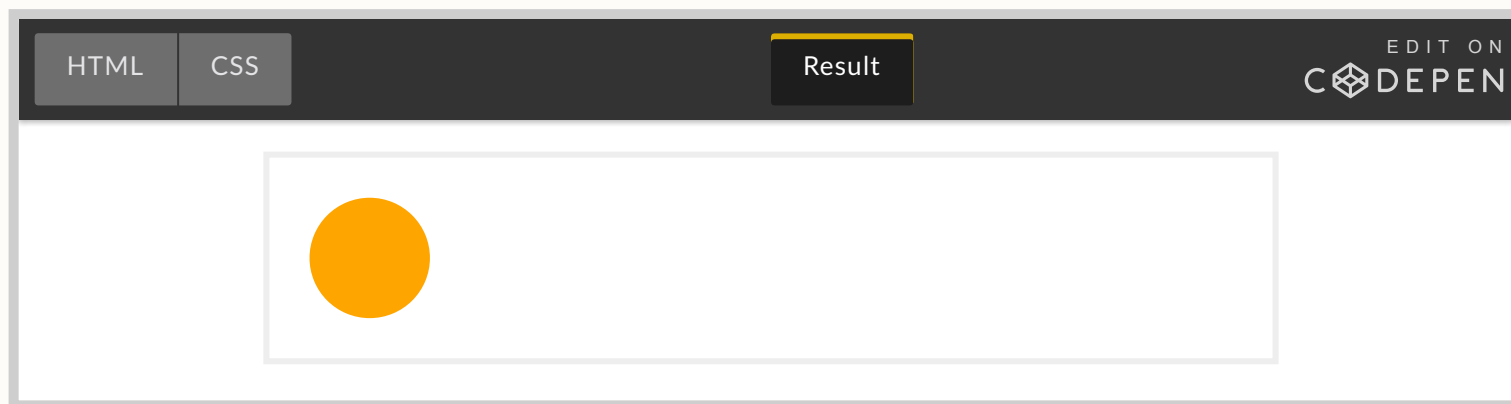
Les attributs `from` et `to` sont similaires aux keyframes `from` et `to` dans un block d'animation CSS `@keyframe` :

```
//CSS
@keyframes moveCircle {
  from { /* valeur de départ */ }
  to { /* valeur d'arrivée */ }
}
```

L'attribut `fill` (qui porte malencontreusement le même nom que l'attribut `fill` définissant la couleur de remplissage d'un élément) est similaire à la propriété `animation-fill-mode` qui spécifie si l'élément devrait ou non revenir à son état initial à la fin de l'animation. Les valeurs en SVG sont similaires à leur équivalent CSS, à part deux noms :

- `freeze` : on indique que l'effet doit rester dans son état final, l'effet d'animation est "gelé" (*freeze*) pour la durée d'existence de la page, ou jusqu'au moment où l'on redémarre l'animation.
- `remove` : l'effet d'animation est retiré (il ne s'applique plus) lorsque la durée de l'animation est passée. L'animation n'affecte plus la cible, sauf si elle est redémarrée.

Essayez de modifier les valeurs dans cette [démonstration CodePen](#) pour voir comment cela affecte l'animation (si vous ne pouvez le faire directement à l'écran, cliquez sur "Edit on CodePen") :



L'attribut `by` est utilisé pour spécifier un décalage de l'animation. Comme son nom le suggère, vous pouvez spécifier le nombre ou la façon par (*by*) lequel vous voulez que l'animation progresse. L'effet de

`by` est surtout visible lorsque vous progressez dans la durée de l'animation en étapes successives, un peu comme avec la fonction CSS `steps()`. L'équivalent SVG de la fonction `steps()` est `calcMode="discrete"`. Nous verrons l'attribut `calcMode` un peu plus loin dans cet article.

Un autre cas où l'effet de `by` est plus visible est quand vous spécifiez uniquement l'attribut `to`. Un exemple serait de l'utiliser avec l'élément `set` que nous allons également voir tout à l'heure.

Et *last but not least*, `by` peut être utile quand vous travaillez avec des animations additives et accumulatives. Plus d'infos tout à l'heure.

Redémarrer les animations avec `restart`

Il peut être utile d'empêcher une animation d'être redémarrée tant qu'elle est active. Pour ce faire, SVG propose l'attribut `restart`. Vous pouvez lui donner l'une des trois valeurs suivantes :

- `always` : l'animation peut être redémarrée n'importe quand. C'est la valeur par défaut.
- `whenNotActive` : l'animation peut être redémarrée seulement quand elle n'est pas active (c'est à dire lorsqu'elle est arrivée à son terme). Les tentatives de redémarrage pendant sa durée d'activité sont ignorées.
- `never` : l'élément ne peut pas être redémarré pour le reste de la durée courante du parent qui la contient. Dans le cas de SVG, puisque le conteneur de temps parent est le fragment de document, l'animation ne peut pas être redémarrée pour le reste de la durée du document.

Nommer les animations et les synchroniser

Supposons que nous voulions animer la position et la couleur du cercle, afin que le changement de couleur se produise à la fin de l'animation de déplacement. Nous pouvons le faire en donnant à la valeur de `begin` de l'animation de changement de couleur la même valeur que la durée de l'animation de déplacement. C'est ce que nous ferions normalement en CSS.

Cependant, SMIL offre une fonctionnalité intéressante de traitement des événements. Nous avons mentionné précédemment que l'attribut `begin` acceptait des valeurs telles que `click + 5s`. Cette valeur est appelée "valeur d'événement", et elle est constituée dans cet exemple d'une référence à l'événement (le clic) suivie d'une "valeur d'horloge". Ce qui est intéressant ici, c'est le nom de la seconde partie: la "valeur d'horloge". Pourquoi pas simplement une "valeur de temps"? Eh bien la réponse est que vous pouvez littéralement utiliser une valeur d'horloge comme "10min" ou "01:33" qui est l'équivalent de 1 minute et 33 secondes, ou même "02:30:03" (deux heures, trente minutes et trois secondes). À l'heure où nous écrivons, les valeurs d'horloge *ne sont implémentées complètement dans aucun navigateur*.

Donc, si nous revenons à la démo précédente et utilisons `click + 01:30`, si un navigateur s'avérait compatible, l'animation serait déclenchée 1 minute 30 après qu'on ait cliqué sur le cercle.

Un autre type de valeur acceptée est l'ID d'une autre animation suivi d'une référence d'événement. Si vous aviez deux animations (ou plus), qu'elles s'appliquent au même élément ou pas, et que vous vouliez les synchroniser de façon à ce que l'une démarre en fonction de l'autre, vous pourriez le faire sans avoir à connaître la durée de l'autre animation.

Par exemple, dans la démo suivante, le rectangle bleu commence à bouger 1 seconde après que l'animation du cercle ait démarré. On donne pour cela une ID à chaque animation, puis on utilise cette ID avec l'événement `begin` comme on le voit dans le code suivant :

```
//SVG
<circle id="orange-circle" r="30" cx="50" cy="50" fill="orange" />

<rect id="blue-rectangle" width="50" height="50" x="25" y="200" fill="#0099cc">
</rect>

<animate
  xlink:href="#orange-circle"
  attributeName="cx"
  from="50"
  to="450"
  dur="5s"
  begin="click"
  fill="freeze"
  id="circ-anim" />

<animate
  xlink:href="#blue-rectangle"
  attributeName="x"
```

```
from="50"  
to="425"  
dur="5s"  
begin="circ-anim.begin + 1s"  
fill="freeze"  
id="rect-anim" />
```

La partie `begin="circ-anim.begin + 1s"` dit au navigateur de démarrer l'animation du rectangle 1 seconde après le début de celle du cercle. Vous pouvez voir l'effet dans cette démo :

HTML

CSS

Result

EDIT ON
CODEPEN

Vous pouvez également démarrer l'animation du rectangle après que celle du cercle soit arrivée à son terme, en utilisant l'événement `end` :


```
//SVG
<animate
  xlink:href="#blue-rectangle"
  attributeName="x"
  from="50"
  to="425"
  dur="5s"
  begin="circ-anim.end"
  fill="freeze"
  id="rect-anim"/>
```

Vous pourriez même la démarrer *avant* la fin de l'animation du cercle :

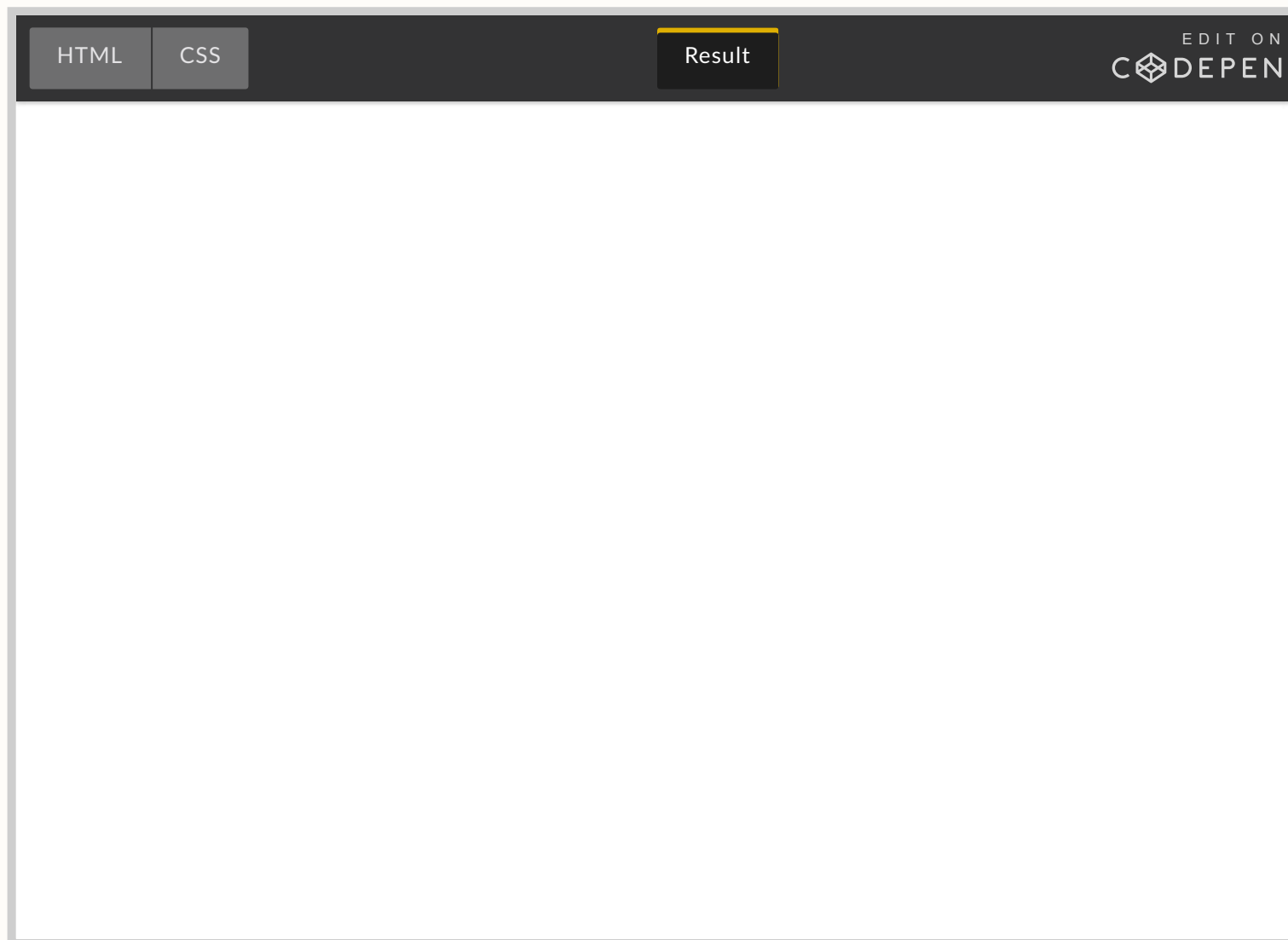
```
//SVG
<animate
  xlink:href="#blue-rectangle"
  attributeName="x"
  from="50"
  to="425"
  dur="5s"
  begin="circ-anim.end - 3s"
  fill="freeze"
  id="rect-anim"/>
```

Répéter les animations

Si vous voulez qu'une animation se produise plus d'une fois, vous pouvez utiliser l'attribut `repeatCount` en spécifiant le nombre de fois que vous voulez répéter l'animation, ou bien utiliser le mot-clé `indefinite` pour qu'elle se répète indéfiniment. Ainsi, si nous voulions que l'animation du cercle se produise deux fois, le code serait :

```
//SVG
<animate
  xlink:href="#orange-circle"
  attributeName="cx"
  from="50"
  to="450"
  dur="5s"
  begin="click"
  repeatCount="2"
  fill="freeze"
  id="circ-anim" />
```

Dans la démo ci-dessous, j'ai donné une valeur de 2 à `repeatCount` sur le cercle, et `indefinite` sur le carré :



Remarquez que l'animation redémarre à partir de la valeur initiale `from` plutôt qu'à partir du point où elle est arrivée. à la fin de l'animation. Malheureusement, SMIL n'offre pas de manière d'aller et venir entre les valeurs de départ et d'arrivée, contrairement à CSS. La propriété `animation-direction` de CSS spécifie si une animation doit repartir en sens inverse sur tous les cycles ou sur certains d'entre eux. La valeur `animate-direction: alternate` indique que les cycles d'animation impairs se produisent dans la direction normale et que les cycles pairs se produisent en sens inverse. Le premier cycle va du début à la fin, le second va de la fin au début, etc.

Dans SMIL, il faudrait utiliser JavaScript pour changer explicitement les valeurs des attributs `from` et `to`. Jon McPartland de Big bite Creative a écrit [un article](#) à ce sujet pour expliquer comment il a créé [une animation d'icône de menu](#).

Une autre façon de contourner le problème serait de spécifier la valeur d'arrivée comme étant une valeur intermédiaire et de faire que la valeur finale soit identique à la valeur initiale. Par exemple, vous pouvez définir une animation commençant `from` une certaine valeur et se terminant à la même valeur, mais en donnant comme valeur intermédiaire entre `from` et `to` la valeur que vous auriez utilisé comme valeur finale.

En CSS, on ferait quelque chose comme cela :

```
//CSS
@keyframes example {
  from, to {
```

```
    left: 0;
  }

  50% {
    left: 300px;
  }
}
```

L'équivalent dans SMIL est d'utiliser l'attribut `values` que nous expliquerons sous peu.

Ceci étant dit, la solution contournée ci-dessus peut fonctionner ou non pour vous selon le type d'animation que vous cherchez et selon que vous enchaînez ou non des animations, que vous les répétiez ou que vous les ajoutiez.

Voci une jolie animation infinie qui utilise des retards dans les débuts d'animation, par Miles Elam :

Restreindre le temps de répétition avec `repeatDur`

Une répétition indéfinie peut s'avérer ennuyeuse ou peu *user-friendly* à la longue, réduire le temps de répétition est donc parfois une bonne solution. C'est ce qu'on appelle le *temps de présentation*.

Le temps de présentation est spécifié à l'aide de l'attribut `repeatDur` dont la syntaxe est similaire à celle de la valeur d'horloge, à ceci près qu'au lieu d'être relatif à un autre événement d'animation ou d'interaction, il est relatif au début du document.

Par exemple, le code suivant stoppera la répétition de l'animation 1 minute et 30 secondes après le début du document :

```
//SVG
<animate
  xlink:href="#orange-circle"
  attributeName="cx"
  from="50"
  to="450"
  dur="2s"
  begin="0s"
  repeatCount="indefinite"
  repeatDur="01:30"
  fill="freeze"
  id="circ-anim" />
```

et voici la démo :

HTML

CSS

Result

EDIT ON
CODEPEN



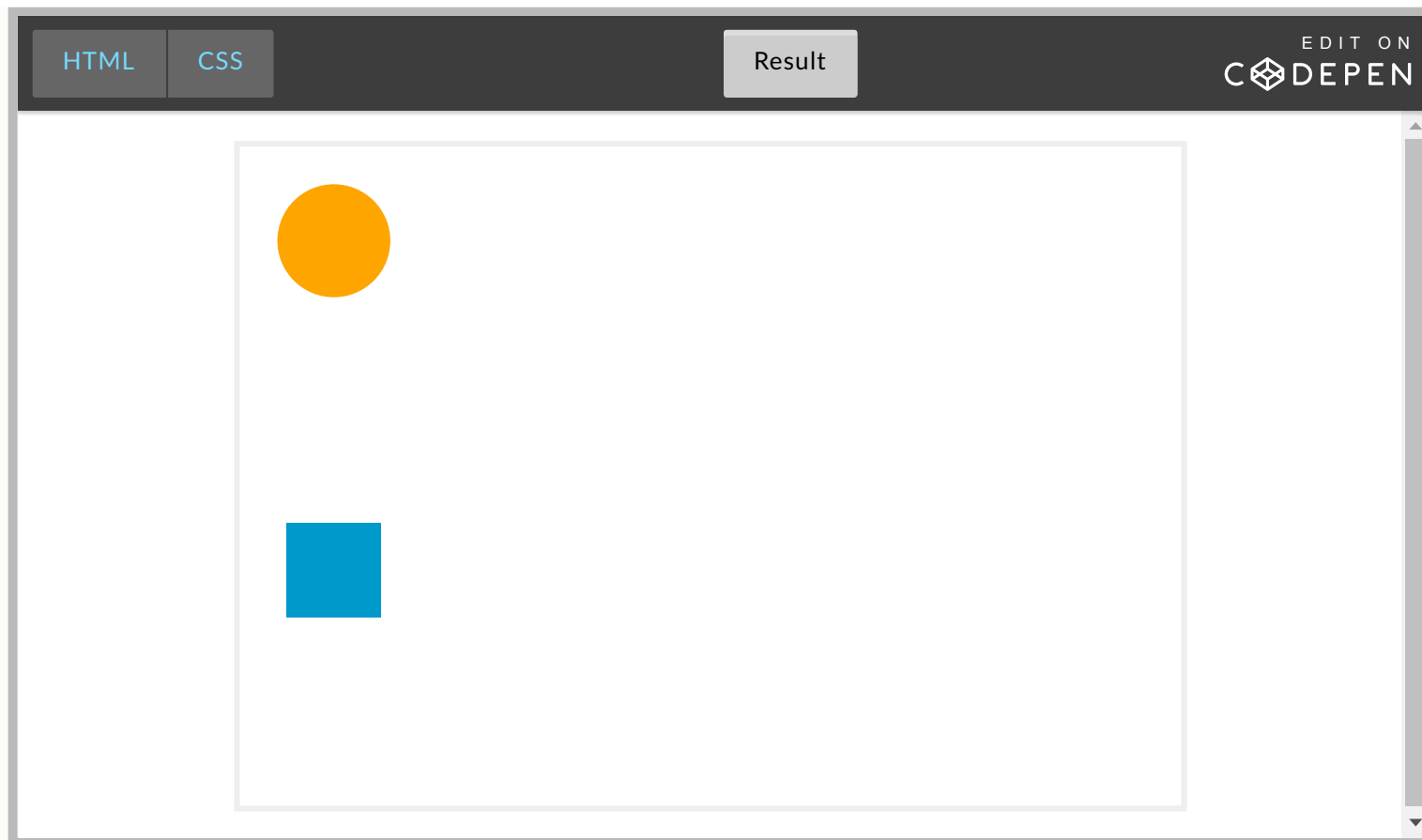
Synchroniser en fonction du nombre de répétitions

Revenons un peu en arrière pour reprendre le sujet de la synchronisation de deux animations. Avec SMIL vous pouvez synchroniser des animations de manière à ce qu'une animation commence en fonction du nombre de répétitions d'une autre. Par exemple, vous pouvez commencer une animation après la n-ième répétition d'une autre animation, plus ou moins une durée que vous pouvez ajouter.

L'exemple qui suit fait débiter l'animation du rectangle à la deuxième répétition de l'animation du cercle :

```
//SVG
<animate
  xlink:href="#blue-rectangle"
  attributeName="x"
  from="50"
  to="425"
  dur="5s"
  begin="circ-anim.repeat(2)"
  fill="freeze"
  id="rect-anim" />
```

Et voici une démo dans laquelle l'animation du rectangle commence 1 seconde après la deuxième répétition de l'animation du cercle.



Et voici un autre exemple, que David Eisenberg a créé pour son livre SVG Essentials, 2e édition.

Contrôler les valeurs de keyframe avec `keyTimes` et `values`

En CSS, nous pouvons spécifier les valeurs que nous voulons donner à notre propriété animée pendant le cours de l'animation. Par exemple, si nous animons le décalage à gauche d'un élément, au lieu de l'animer de 0 à 300 directement, nous pouvons l'animer de façon à ce qu'il prenne certaines valeurs pendant certaines périodes de temps :

```
//CSS
@keyframes example {
  0% {
    left: 0;
  }
  50% {
    left: 320px;
  }
  80% {
    left: 270px;
  }
  100% {
    left: 300px;
  }
}
```

0%, 50%, 80% et 100% sont les keyframes (étapes) de l'animation et les valeurs comprises dans chaque bloc sont celles de chaque keyframe. L'effet décrit ci-dessus est celui d'un élément qui

rebondit contre un mur et revient à la position finale.

Dans SMIL, vous pouvez contrôler les valeurs par étape de la même façon, mais la syntaxe est différente.

Pour spécifier les keyframes, on utilise l'attribut `keyTimes`. Puis pour spécifier la valeur de la propriété animée à chaque étape, on utilise les attributs `values`. Les conventions de nommage de SMIL sont très pratiques.

Si je reviens à notre cercle et que j'utilise des valeurs similaires à celles de l'exemple CSS précédent, le code ressemblera à ceci :

```
//SVG
<animate
  xlink:href="#orange-circle"
  attributeName="cx"
  from="50"
  to="450"
  dur="2s"
  begin="click"
  values="50; 490; 350; 450"
  keyTimes="0; 0.5; 0.8; 1"
  fill="freeze"
  id="circ-anim" />
```

Qu'avons-nous fait ici ?

La première chose à remarquer est que les temps et valeurs intermédiaires des keyframes sont spécifiés sous forme de liste. L'attribut `keyTimes` est une liste de valeurs temporelles séparées par des points-virgules, utilisée pour contrôler l'avancée de l'animation. Chaque temps dans la liste correspond à une valeur dans la liste de l'attribut `values` et définit le moment où la valeur est utilisée dans la fonction d'animation. Chaque valeur temporelle dans la liste `keyTimes` est comprise entre 0 et 1 (compris), la différence avec CSS est donc qu'au lieu d'être indiquée en pourcentage, elle l'est sous forme de fraction.

Voici la démo du code précédent. Cliquez sur le cercle pour démarrer l'animation.

HTML

CSS

Result

EDIT ON
CODEPEN

Remarquez que si on utilise une liste de valeurs, l'animation appliquera les valeurs dans l'ordre pendant le cours de l'animation. De plus, si une liste de `values` est spécifiée, toute valeur d'attributs `from`, `to` et `by` est ignorée.

Autre chose à savoir : vous pouvez utiliser l'attribut `values` sans l'attribut `keyTimes` – les valeurs sont automatiquement espacées de manière régulière dans le temps (pour chaque valeur `calcMode` différente de `paced`, voir section suivante).

Contrôler la vitesse d'animation avec un easing personnalisé, `calcMode` et `keySplines`

Je vais comparer à nouveau SMIL et CSS parce qu'il est plus facile de comparer les syntaxes et les concepts lorsqu'on connaît déjà les animations CSS.

En CSS, vous pouvez choisir de modifier le rythme de l'animation, uniforme par défaut, et de spécifier une fonction d'easing personnalisée qui contrôle l'animation, grâce à la propriété `animation-timing-function`. La fonction `timing` peut être l'un des mots-clés prédéfinis ou une courbe de Bézier cubique. Cette dernière peut être créée via un outil tel que celui proposé par Lea Verou.

Dans SMIL, le rythme de l'animation est spécifié avec l'attribut `calcMode`. Par défaut, l'animation est linéaire pour tous les éléments, à l'exception de `animateMotion` (que nous verrons tout à l'heure). Outre la valeur `linear`, vous pouvez donner une valeur de `discrete`, `paced` ou `spline`.

- `discrete` spécifie que l'animation sautera d'une valeur à l'autre sans interpolation. C'est la même chose que la fonction `steps()` en CSS.

- `paced` est similaire à `linear`, mais il ignorera tout les temps intermédiaires définis par `keyTimes`. Il calcule la distance entre les valeurs consécutives et divise le temps en fonction. Si vos valeurs sont toutes en ordre linéaire, vous ne remarquerez pas la différence. Mais si elles vont en avant et en arrière, ou si ce sont des couleurs (qui sont traitées comme des valeurs vectorielles tri-dimensionnelles), vous verrez les valeurs intermédiaires. Ci-dessous, voici une animation créée par Amelia Bellamy-Royds qui montre la différence entre les trois valeurs `calcMode` mentionnées jusqu'ici.

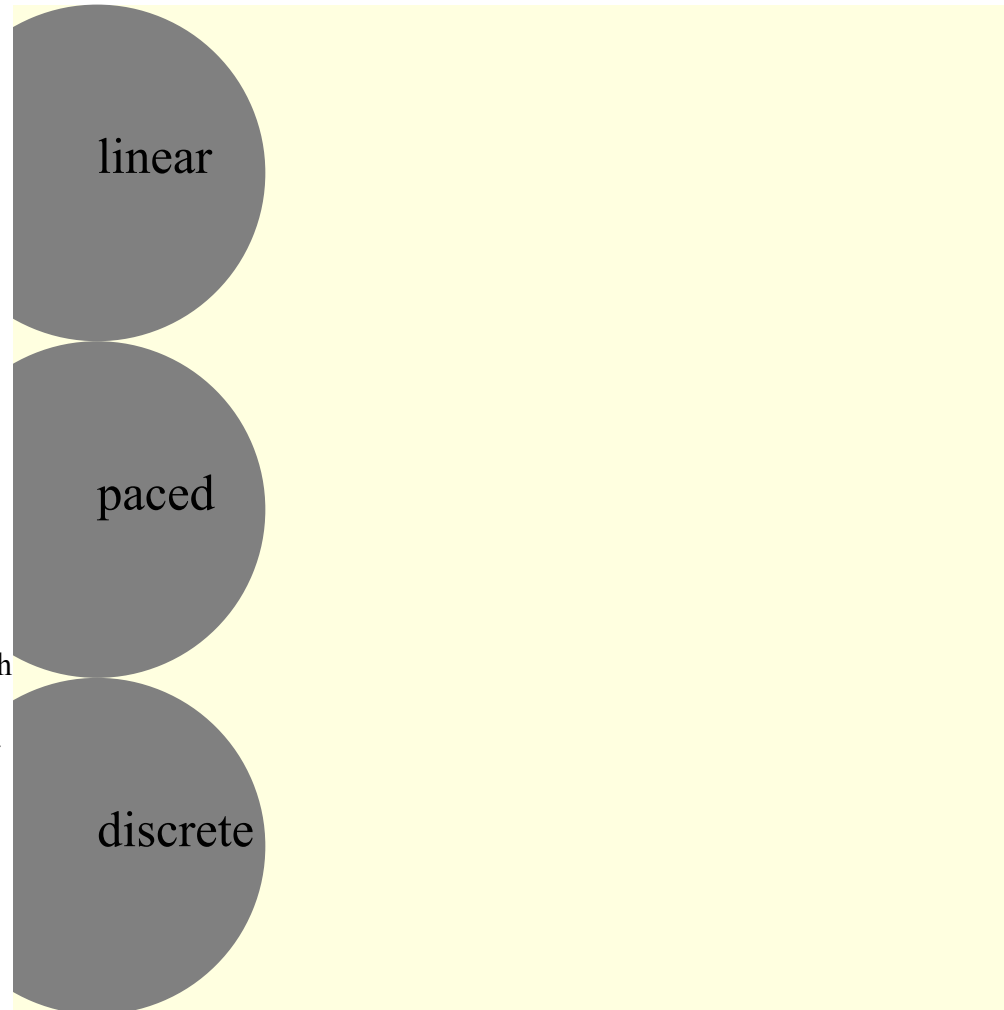
Linear vs. Paced vs. Discrete SVG/SMIL Animation

Click the SVG to restart the animations.

The SVG/SMIL `calcMode` attribute on animation elements defines how to interpolate between multiple values in a single animation.

Linear animation (the default) divides up the time evenly between multiple specified values, and then progresses between each stop point in an even manner. So if you have 5 values, the time will be divided into 4 transitions between each successive pair of values. You can specify different timings with a `keyTimes` attribute, but each step will still interpolate linearly. For `keyTimes`, use another semi-colon separated list with the same number of entries as your values list. The first time must be 0 and the last time 1; other times are given as decimals (representing a proportion of total time), in numerical order.

Paced animation calculates the



Linear animation calculates the distance between each values, and divides up the time accordingly to create an even speed of change throughout the

- La quatrième valeur acceptée par `calcMode` est `spline`. Elle interpole d'une valeur à l'autre de la liste des `values` selon une fonction temporelle définie par une courbe de bézier cubique. Les points sur la spline sont définis dans l'attribut `keyTimes` et les points de contrôle pour chaque intervalle sont définis dans l'attribut `keySplines`.

📖 *NdT* : pour mieux comprendre les keysplines, vous pouvez consulter cette page sur [l'interpolation spline](#), ou celle-ci sur [keySplines](#). Vous pouvez également regarder la vidéo sur cette [présentation des courbes de bézier](#) et constater la présence de `keySplines` dans le codePen qui suit.

Vous avez remarqué un nouvel attribut dans la dernière phrase : `keySplines`. À quoi sert-il ?

Là encore, reprenons les équivalences CSS.

En CSS, vous pouvez spécifier le rythme de l'animation à *l'intérieur* de chaque keyframe, au lieu de le spécifier pour toute l'animation. Cela vous donne un meilleur contrôle de chaque keyframe. Un bon exemple de cette fonctionnalité est l'effet de la balle qui rebondit. Les keyframes ressembleraient à ceci :

```
//CSS
@keyframes bounce {
  0% {
    top: 0;
    animation-timing-function: ease-in;
  }
  15% {
    top: 200px;
    animation-timing-function: ease-out;
  }
  30% {
    top: 70px;
    animation-timing-function: ease-in;
  }
  45% {
    top: 200px;
    animation-timing-function: ease-out;
  }
  60% {
    top: 120px;
    animation-timing-function: ease-in;
  }
  75% {
```

```
    top: 200px;
    animation-timing-function: ease-out;
}
90% {
    top: 170px;
    animation-timing-function: ease-in;
}
100% {
    top: 200px;
    animation-timing-function: ease-out;
}
```

```
}
```

À la place des mots-clés définissant les fonctions d'easing, nous aurions pu utiliser les courbes de bézier correspondantes :

- `ease-in` = `cubic-bezier(0.47, 0, 0.745, 0.715)`
- `ease-out` = `cubic-bezier(0.39, 0.575, 0.565, 1)`

Commençons en spécifiant les `keyTimes` et la liste de `values` pour donner à notre cercle orange le même effet rebondissant.

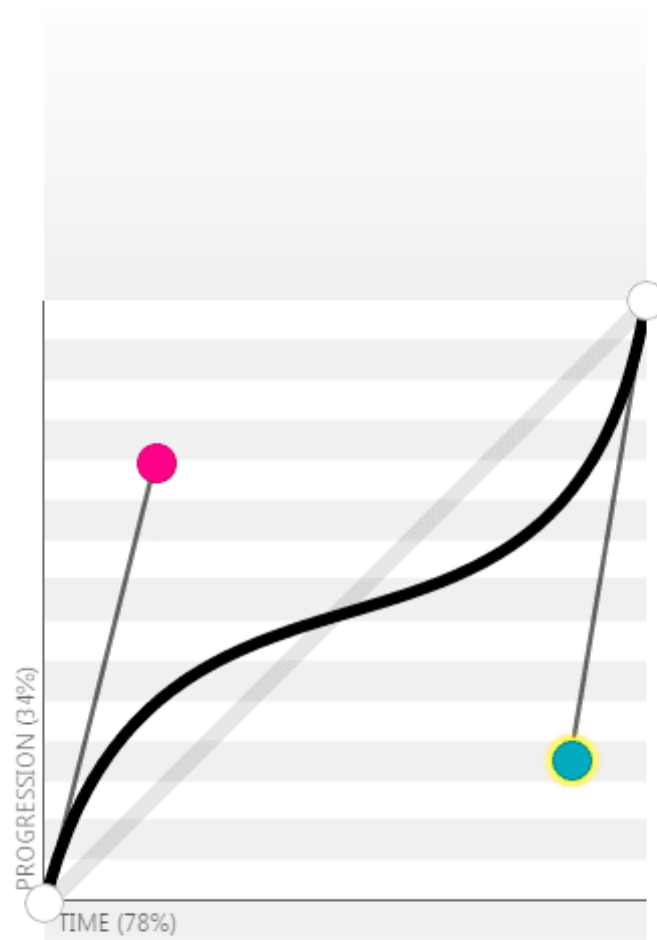
```
//SVG
<animate
  xlink:href="#orange-circle"
  attributeName="cy"
  from="50"
  to="250"
  dur="3s"
  begin="click"
  values="50; 250; 120;250; 170; 250; 210; 250"
  keyTimes="0; 0.15; 0.3; 0.45; 0.6; 0.75; 0.9; 1"
  fill="freeze"
  id="circ-anim" />
```

L'animation commencera au clic et s'arrêtera une fois atteinte la valeur finale. Ensuite, pour spécifier le rythme de chaque keyframe, nous allons ajouter l'attribut `keySplines`.

L'attribut `keySplines` accepte un ensemble de points de contrôle de Bézier associés avec la liste `keyTimes`, qui définissent une fonction cubique de Bézier contrôlant l'allure de chaque intervalle. La valeur de l'attribut est une liste de points de contrôles séparés par un point-virgule. Chaque point de contrôle est décrit par un ensemble de quatre valeurs : $x_1 y_1 x_2 y_2$, représentant les points de contrôle Bézier pour un segment temporel. Les valeurs doivent être comprises entre 0 et 1 et l'attribut est ignoré si le `calcMode` n'est pas réglé sur `spline`.

Plutôt que de prendre des fonctions de Bézier comme valeurs, les `keySplines` prennent les coordonnées des deux points de contrôle utilisés pour dessiner la courbe. Les points de contrôle peuvent être vus sur ces captures d'écran prises sur le site de Lea Verou. On peut voir les coordonnées de chaque point, coloriées de la même façon que le point lui-même. Ce sont ces valeurs que nous allons utiliser avec l'attribut `keySplines` pour définir l'allure des animations keyframe.

Dans SMIL ces valeurs peuvent être séparées par des virgules ou par un espace. Les valeurs `keyTimes` qui définissent le segment associé sont les "points d'ancrage" Bézier, et les valeurs `keySplines` sont les points de contrôle. Par conséquent, il doit y avoir un ensemble de points de contrôle *en moins* par rapport au nombre de `keyTimes`.

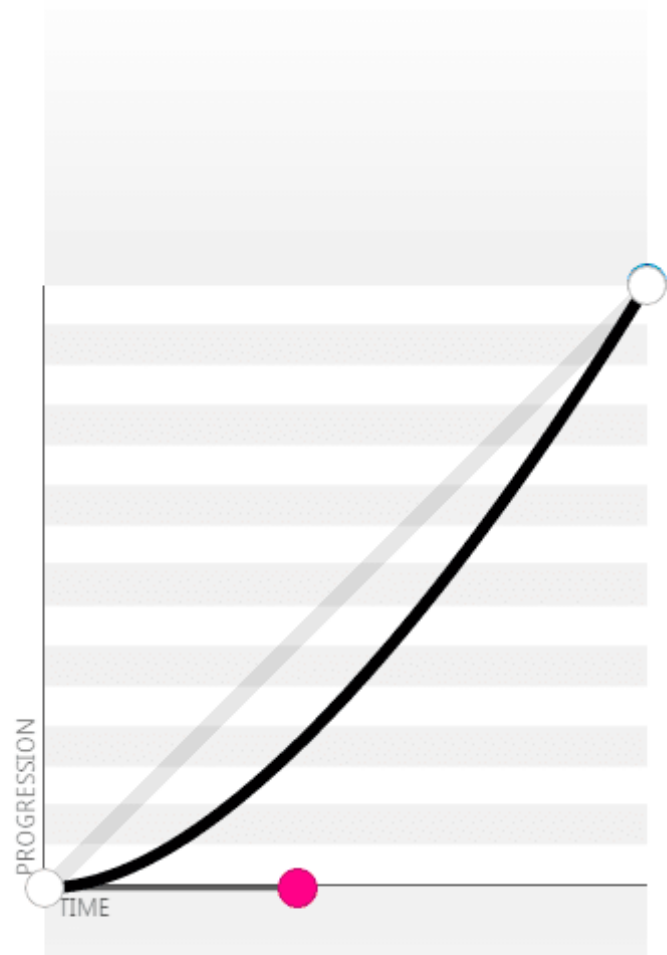


cubic-bezier(.18, .73, .87, .24)

Si nous revenons à notre exemple de ballon qui rebondit, les coordonnées des points de contrôle pour les fonctions d'ease-in et ease-out apparaissent dans les images suivantes :

DONATE

Made by [Lea Verou](#) with care [About](#)

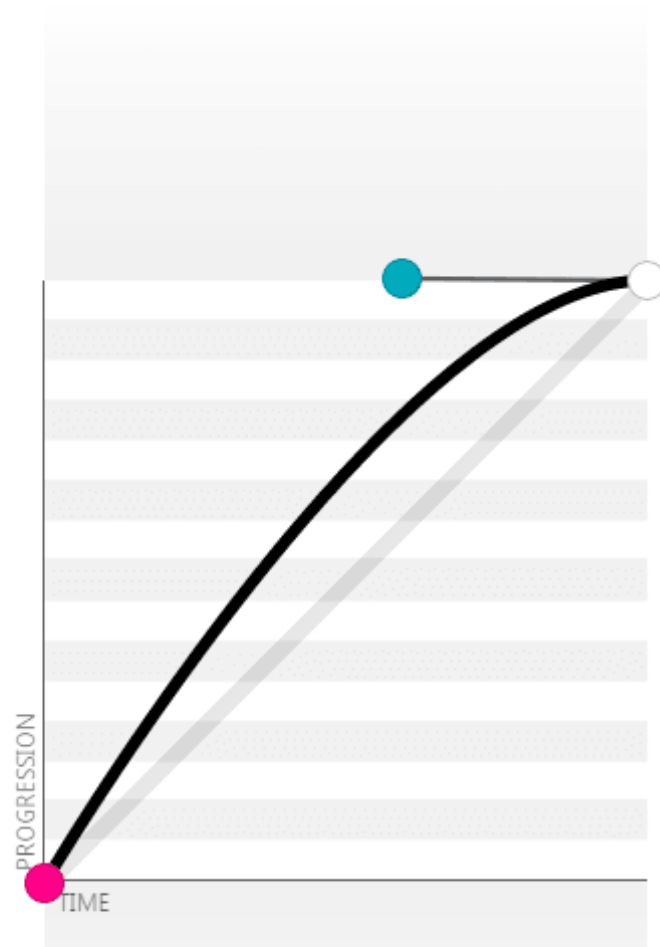


`cubic-bezier(.42,0,1,1)`

DONA

DONATE

Made by [Lea Verou](#) with care  [About](#)



cubic-bezier(0,0,.59,1)

Pour traduire cela en élément d'animation SVG, nous utilisons le code suivant :

```
//SVG
<animate
  xlink:href="#orange-circle"
  attributeName="cy"
  from="50"
  to="250"
  dur="3s"
  begin="click"
  values="50; 250; 120; 250; 170; 250; 210; 250"
  keyTimes="0; 0.15; 0.3; 0.45; 0.6; 0.75; 0.9; 1"
  keySplines=".42 0 1 1;
             0 0 .59 1;
             .42 0 1 1;
             0 0 .59 1;
             .42 0 1 1;
             0 0 .59 1;
             .42 0 1 1;"
  fill="freeze"
  id="circ-anim"/>
```

Voici la démo :

HTML

CSS

Result

EDIT ON
CODEPEN



Click on the circle to start the animation.

Si vous ne voulez spécifier qu'une fonction d'easing pour l'animation entière, sans valeurs intermédiaires, il vous faut quand même spécifier les keyframes en utilisant l'attribut `keyTimes`, mais

vous n'indiquez que les keyframes de début et de fin, c'est à dire `0; 1` et aucune `values` intermédiaire.

Ajouter et accumuler des animations

Parfois il peut être utile de définir qu'une animation commence là où la précédente s'est achevée. Ou bien de définir qu'une animation utilise les valeurs accumulées des animations précédentes comme valeur à partir de laquelle poursuivre. Pour cela, SVG a deux attributs bien nommés : `additive` et `accumulate`.

Supposons que nous ayons un élément que nous voulons élargir, ou une ligne que nous souhaitons allonger, ou un élément que nous voulons faire évoluer pas à pas d'une position à une autre. Cette fonctionnalité est particulièrement utile pour les animations répétées.

Comme pour toute autre animation, nous allons spécifier les valeurs `from` et `to`. Cependant, lorsque nous réglons la valeur d'`additive` sur `sum`, les valeurs de `from` et `to` seront relatives à la valeur originale de l'attribut animé. Si nous revenons à notre cercle, la position initiale de `cx` est 50. Lorsque nous fixons `from="0" to="100"`, le point de départ est donc 50 et le point d'arrivée est 100+50, c'est donc en pratique comme nous écrivions "`from="50" to="150"`".

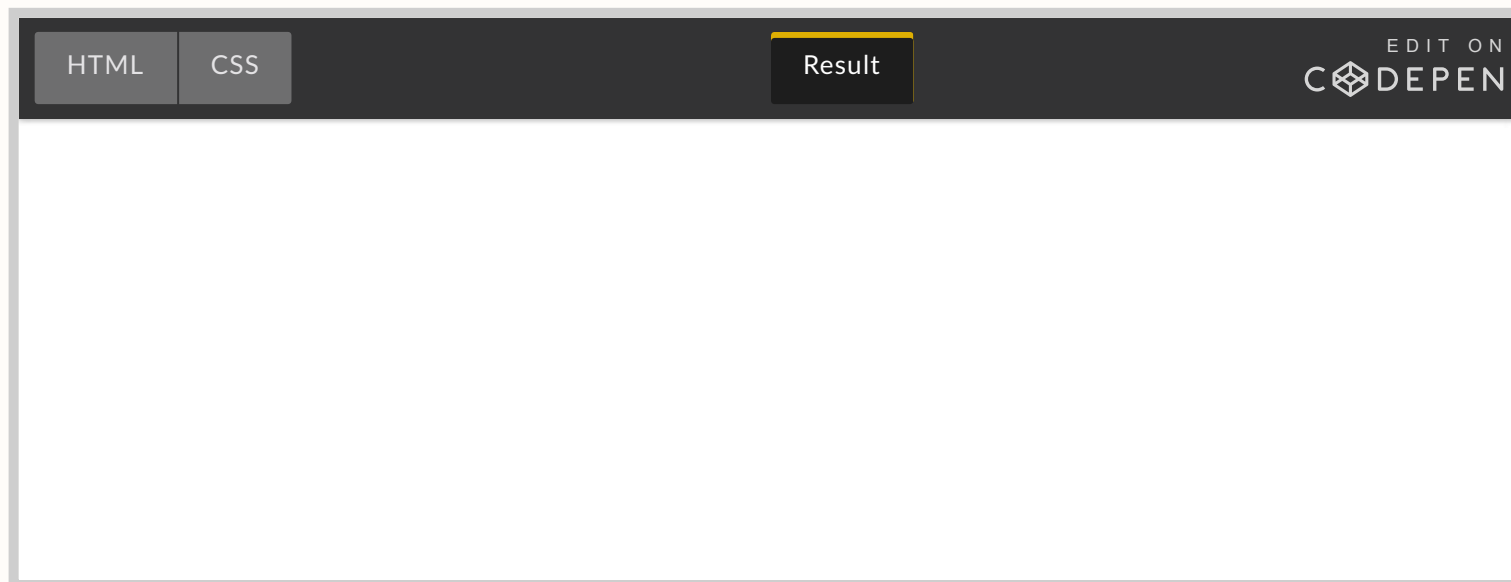
Nous obtenons le résultat suivant :

C'est tout ce que fait l'attribut `additive`. Il indique simplement si les valeurs `from` et `to` doivent être relatives à la valeur courante (ici, de `cx`) ou non. L'attribut prend l'une des deux valeurs suivantes : `sum` ou `replace`. Cette dernière est la valeur par défaut et elle signifie que les valeurs de `from` et `to` remplaceront les valeurs courantes ou originales – ce qui peut causer un saut bizarre juste avant le début de l'animation (faites l'expérience en remplaçant `sum` par `replace` dans l'exemple précédent).

Mais comment faire si nous voulons que les valeurs soient additionnées de manière telle que la seconde animation débute au point d'arrivée de la première ? C'est ici qu'intervient l'attribut `accumulate`.

L'attribut `accumulative` contrôle si l'animation est, ou non, cumulative. La valeur par défaut est `none`, ce qui signifie que par exemple lorsque l'animation est répétée elle recommence depuis le début. Vous pouvez la régler sur `sum`, qui spécifie que chaque répétition repart de la dernière valeur.

Si nous reprenons notre animation, `accumulate="sum"` donnera le résultat suivant :

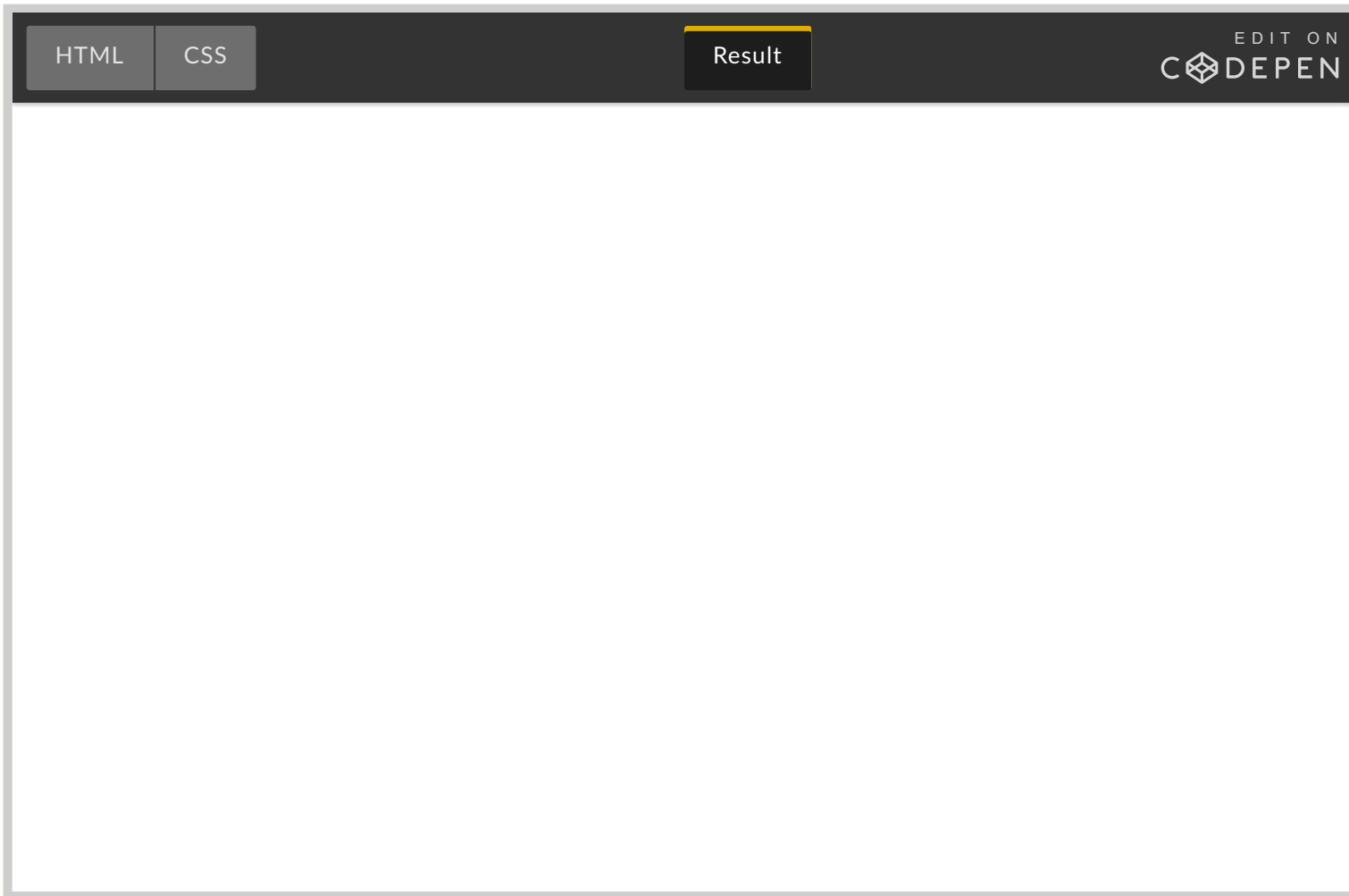


L'attribut `accumulate` est ignoré si la valeur d'attribut cible n'est pas additionnable ou si l'élément d'animation ne se répète pas. Il sera également ignoré si la fonction d'animation est définie uniquement avec l'attribut `to`.

Spécifier la fin de l'animation avec `end`

En plus de définir le début d'une animation, on peut définir sa fin, avec l'attribut `end`. Par exemple, nous pouvons déterminer qu'une animation se répètera indéfiniment, puis la faire cesser lorsqu'un autre élément débute son animation. L'attribut `end` accepte des valeurs similaires à celles de `begin`. On peut spécifier des valeurs absolues ou relatives pour le temps, les répétitions, les événements, etc.

Par exemple, dans la démo qui suit, le cercle orange se déplace lentement sur une période de 30 secondes. Le cercle vert s'anamera quand on clique dessus, et l'animation du cercle orange cessera au moment où débutera celle du cercle vert. Cliquez sur le cercle vert pour voir le cercle orange s'arrêter :



On peut évidemment réaliser le même genre de synchronisation d'animations lorsqu'il s'agit de deux animations appliquées au même élément. Par exemple, supposons que nous réglions la couleur du cercle de façon à ce qu'elle s'anime indéfiniment en passant d'une valeur à une autre. Puis, lorsqu'on clique sur l'élément il se déplace vers l'autre côté. Nous voulons que l'animation couleur s'arrête dès qu'on clique sur l'élément et que l'animation de déplacement est lancée.

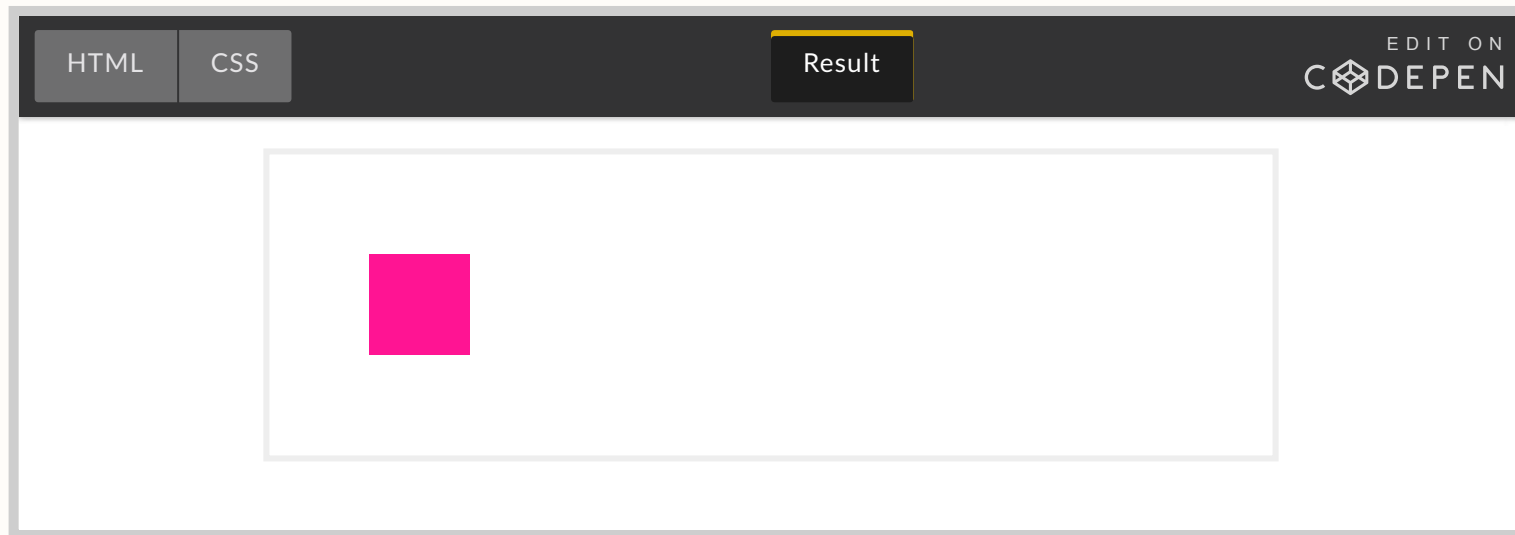
Définir des intervalles d'animation avec plusieurs `begin` et `end`

En fait, les attributs `begin` et `end` acceptent tous les deux une liste de valeurs séparées par un point-virgule. Chaque valeur dans l'attribut `begin` correspondra à une valeur dans l'attribut `end`, formant ainsi des intervalles d'animation actifs et inactifs.

On peut le voir comme une voiture qui se déplace, avec les roues de la voiture qui seraient actives et inactives pendant certaines périodes de temps, selon que la voiture bouge ou pas. Vous pouvez même créer l'effet de la voiture animée en appliquant deux animations à la voiture : l'une qui déplace la

voiture le long d'un chemin qui est aussi une animation additive et accumulative, et l'autre animation qui fait tourner les roues de la voiture dans des intervalles synchronisés avec le déplacement.

La démo suivante est un exemple de débuts et fins multiples (c'est à dire d'intervalles multiples), avec un rectangle qui tourne sur lui-même à intervalles déterminés, passant d'un état actif à inactif (faites rerun si l'animation s'est terminée).



Remarquez que dans cet exemple j'ai utilisé l'élément `<animateTransform>` pour faire tourner le rectangle sur son centre. Nous allons revenir sur cet élément plus en détail tout à l'heure.

Remarquez également que même si vous réglez `repeatCount` sur `indefinite` les valeurs `end` l'emporteront et l'animation ne se répètera pas indéfiniment.

Restreindre la durée d'activité d'un élément avec `min` et `max`

De même qu'on peut restreindre le nombre de répétitions d'une animation, on peut restreindre la **durée active** d'une animation. Les attributs `min` et `max` spécifient la valeur minimum et maximum de la durée active. On a ainsi un moyen de contrôler les limites inférieures et supérieures de la durée active de l'élément. Chacun de ces attributs prend une valeur de type valeur d'horloge.

Pour `min`, cela spécifie la longueur de la valeur minimum de la durée active. La valeur doit être supérieure ou égale à zéro, qui est la valeur par défaut et ne contraint pas la durée active.

Pour `max`, la valeur d'horloge spécifie la longueur de la valeur maximum de la durée active. Elle doit également être supérieure à zéro. La valeur par défaut de `max` est `indefinite`, elle ne contraint pas la durée active.

Si les attributs `min` et `max` sont tous les deux spécifiés, la valeur de `max` doit être supérieure ou égale à la valeur de `min`, faute de quoi les deux attributs sont ignorés.

Mais qu'est-ce qui définit la durée active d'un élément ? Nous avons déjà mentionné la durée de répétition, en plus de la "simple durée" qui est la durée de l'animation sans répétition (spécifiée avec `dur`), alors comment toutes ces durées fonctionnent-elles ensemble ? Laquelle prend le dessus ? Et où intervient l'attribut `end` qui prendrait le pas sur tous les autres pour mettre fin à l'animation ?

Les choses se passent de la manière suivante : le navigateur va *d'abord* calculer la durée active en fonction des valeurs `dur`, `repeatCount`, `repeatDur` et `end`. Puis, il compare cette durée calculée avec les valeurs `min` et `max`. Si le résultat est à l'intérieur des limites, cette première durée calculée est correcte et ne sera pas modifiée. Sinon, deux situations sont possibles :

- Si la première durée calculée est supérieure à la valeur `max`, la durée active de l'élément est définie comme égale à `max`.
- Si la première durée calculée est inférieure à la valeur `min`, la durée active de l'élément est définie comme égale à `min` et l'élément se comporte comme suit :
 - Si la durée de répétitions (ou la durée simple, si l'élément ne se répète pas) de l'élément est supérieure à `min`, alors l'élément est animé normalement pendant la durée active (avec la contrainte `min`).
 - Sinon, l'élément est animé normalement pour sa durée de répétition (ou sa durée simple s'il ne se répète pas) et il est arrêté (gelé) ou il n'est pas montré selon la valeur de l'attribut `fill`.

Il nous reste maintenant à voir comment le navigateur calcule la durée active. Pour faire bref, je n'entrerai pas dans les détails ici. Mais vous pouvez trouver dans [la spécification](#) un tableau complet détaillant les combinaisons de `dur`, `repeatCount`, `repeatDur` et `end` et ce que deviendra la durée active en fonction de la combinaison de ces attributs.

Enfin, si on définit qu'un élément doit commencer avant son parent (par exemple avec une valeur de décalage négative), la durée minimum est mesurée à partir du temps calculé de départ, et non à partir

du temps observé. Cela signifie que la valeur `min` peut n'avoir aucun effet observé.

Un exemple d'`animate` : morphing des chemins

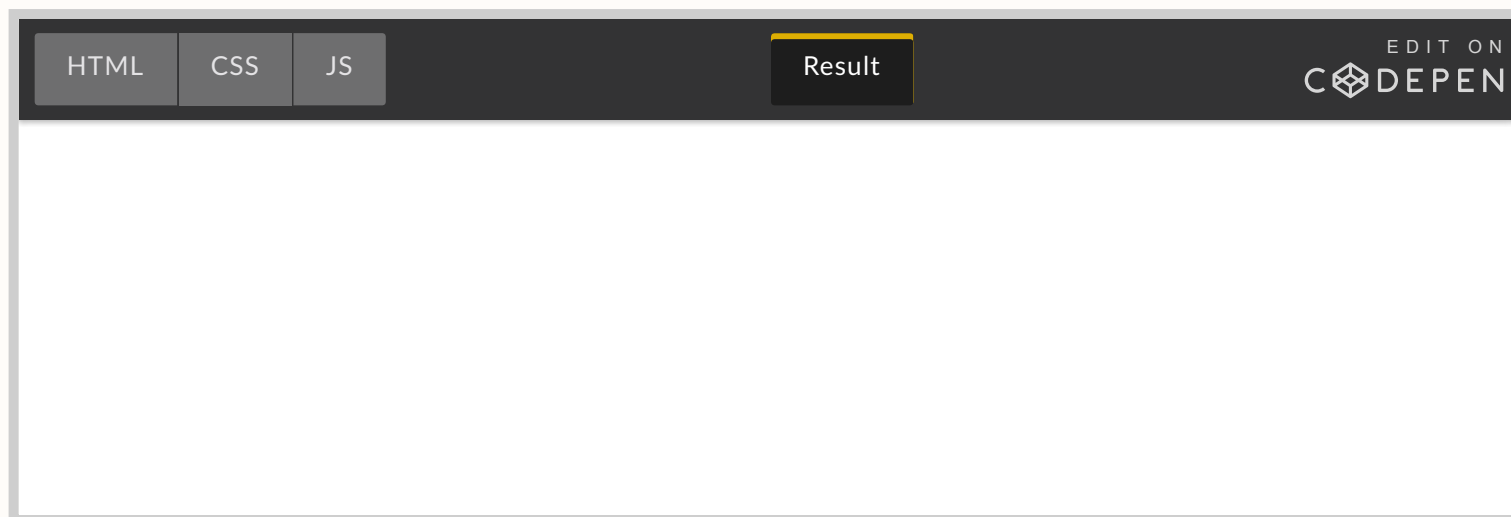
Un des attributs qu'on peut animer en SMIL (mais pas en CSS) est l'attribut `d` (raccourci pour *data*) d'un `<path>` SVG. L'attribut `d` contient les données définissant le contour de la forme que nous dessinons. Elles sont constituées par un ensemble de commandes et de coordonnées qui indiquent au navigateur où et comment dessiner des points, des arcs, des lignes qui forment le chemin final.

L'animation de cet attribut nous permet de *morpher* les chemins SVG et de créer des effets d'interpolation de formes. Mais pour pouvoir réaliser ce morphing, les chemins de début, de fin et tous les chemins intermédiaires doivent avoir le même nombre de sommets et de points, qui doivent apparaître dans le même ordre. Si le nombre de sommets ne correspond pas, l'animation ne marchera pas. La raison est que les modifications de la forme sont produites par le déplacement des sommets et l'interpolation de leur positions.

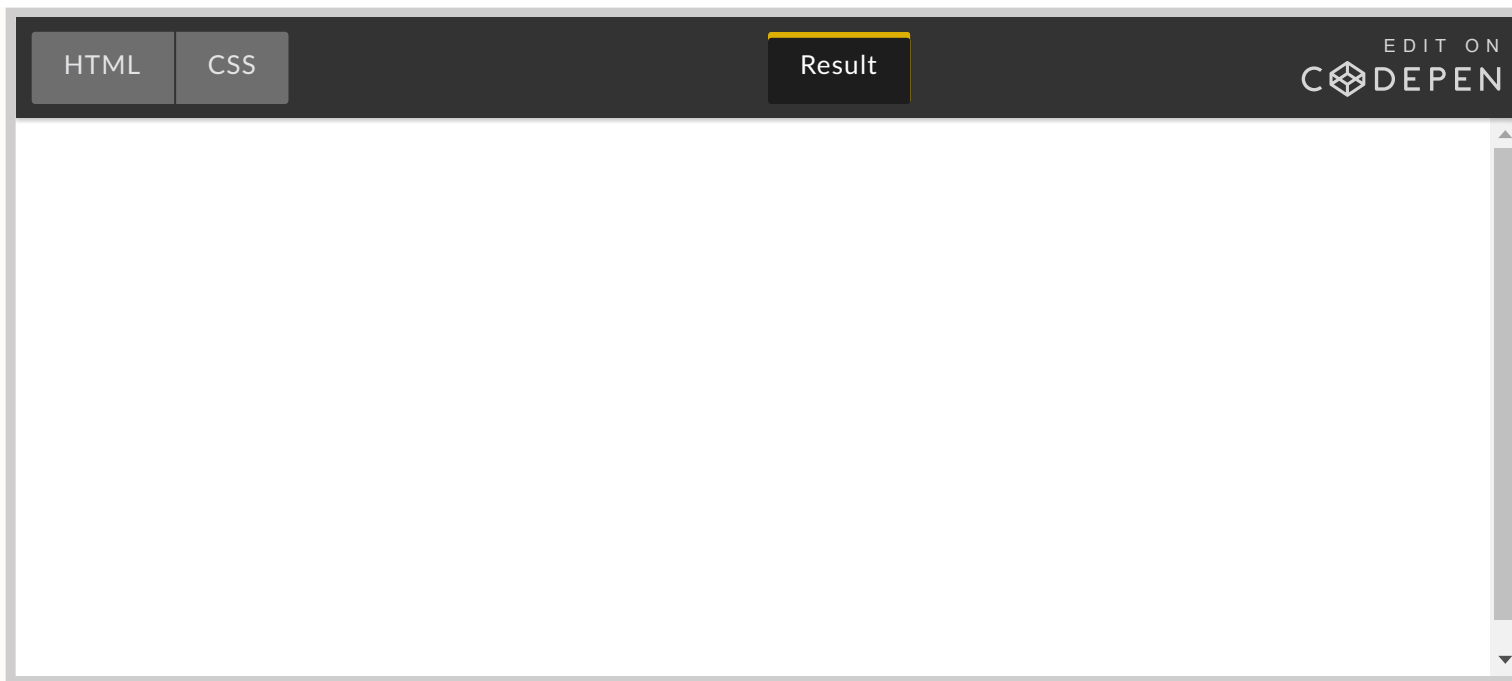
Pour animer un chemin SVG, on spécifie que l'`attributeName` doit être `d`, et on fixe les valeurs de `from` et `to` qui indiquent les formes de début et de fin, et on peut utiliser l'attribut `values` pour indiquer toute valeur intermédiaire.

Là non plus je n'entrerai pas dans les détails. Vous pouvez lire [cet excellent article de Noah Blon](#) dans lequel il explique comment il a créé une animation utilisant l'interpolation de formes avec `<animate>`.

La démo live de l'article de Noah ressemble à ceci :



Et voici un autre exemple de morphing par Felix Hornoiu :



Vous pouvez même morpher les valeurs d'un chemin utilisé comme masque de détournage ! En voici un exemple par Heather Buchel :

Animer le long de chemins arbitraires avec `animateMotion`

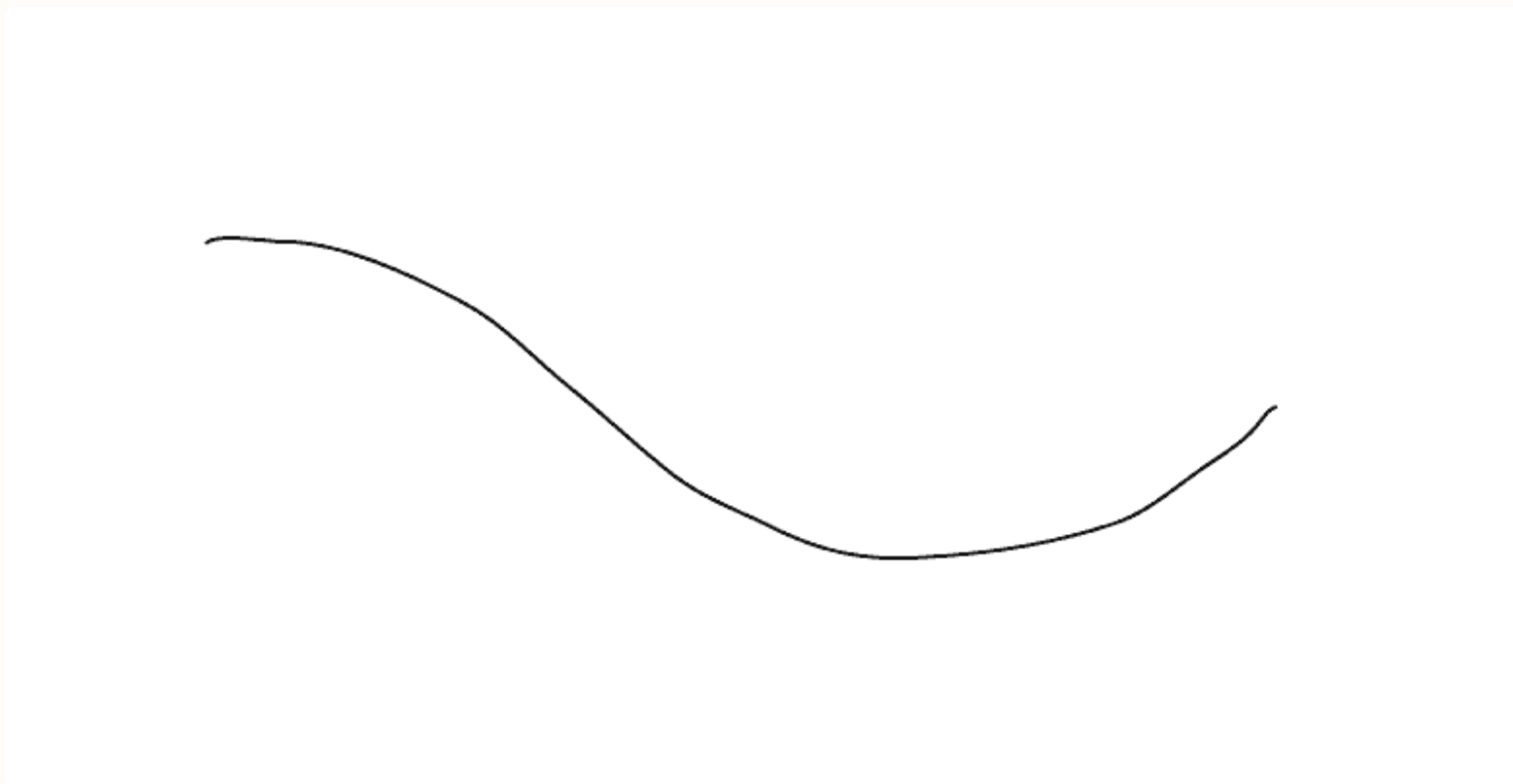
L'élément `<animateMotion>` est mon élément SMIL préféré. Vous pouvez l'utiliser pour déplacer un élément le long d'un chemin. On peut spécifier le chemin de déplacement de deux façons possibles, que nous allons voir tout à l'heure.

L'élément `<animateMotion>` accepte les mêmes attributs que mentionnés précédemment, plus trois autres : `keyPoints`, `rotate` et `path`. Par ailleurs, concernant l'attribut `calcMode`, la valeur par défaut est `paced` et non `linear`.

Spécifier le chemin avec l'attribut `path`

L'attribut `path` est utilisé pour spécifier le chemin de déplacement. Il est exprimé dans le même format et interprété de la même façon que l'attribut `d` sur l'élément `path`.

Nous allons animer notre cercle le long d'un chemin qui ressemble à ceci :



Voici le code nécessaire pour déplacer le cercle le long de ce chemin :

```
//SVG
<animateMotion
  xlink:href="#circle"
  dur="1s"
  begin="click"
  fill="freeze"
  path="M0,0c3.2-3.4,18.4-0.6,23.4-0.6c5.7,0.1,10.8,0.9,16.3,2.3
c13.5,3.5,26.1,9.6,38.5,16.2c12.3,6.5,21.3,16.8,31.9,25.4
  c10.8,8.7,21,18.3,31.7,26.9c9.3,7.4,20.9,11.5,31.4,16.7
  c13.7,6.8,26.8,9.7,41.8,9c21.4-1,40.8-3.7,61.3-10.4
  c10.9-3.5,18.9-11.3,28.5-17.8c5.4-3.7,10.4-6.7,14.8-11.5
  c1.9-2.1,3.7-5.5,6.5-6.5" />
```

Je voudrais attirer votre attention sur un point : les coordonnées à l'intérieur de path. Le chemin commence en se mouvant(**M**) vers le point de coordonnées (0,0), avant de commencer à dessiner une courbe (**c**) vers un autre point. Il est important de noter que le point (0,0) est en fait la position du cercle, où qu'il se trouve, et non l'angle supérieur gauche du système de coordonnées. Les coordonnées à l'intérieur de l'attribut `path` sont relatives à la position *actuelle* de l'élément !

Le résultat est le suivant :

HTML

CSS

Result

EDIT ON
CODEPEN



Click on the circle to animate it.

Si vous spécifiez le chemin en partant d'un point autre que (0,0), le cercle sauterait brusquement de sa position actuelle à la position spécifiée. Imaginez que vous dessiniez un chemin dans Illustrator

puis que vous exportiez ces données de chemin pour les utiliser comme chemin de déplacement (c'est ce que j'ai fait la première fois...) le chemin exporté pourrait ressembler à ceci :

```
//SVG
<path fill="none" stroke="#000000" stroke-miterlimit="10" d="M100.4,102.2c3.2-
3.4,18.4-0.6,23.4-0.6c5.7,0.1,10.8,0.9,16.3,2.3
c13.5,3.5,26.1,9.6,38.5,16.2c12.3,6.5,21.3,16.8,31.9,25.4c10.8,8.7,21,18.3,31.7,26.9c9.
c13.7,6.8,26.8,9.7,41.8,9c21.4-1,40.8-3.7,61.3-10.4c10.9-3.5,18.9-11.3,28.5-17.8c5.4-
3.7,10.4-6.7,14.8-11.5
c1.9-2.1,3.7-5.5,6.5-6.5"/>
```

Dans ce cas, le point de départ est (100.4, 102.2) et si nous utilisons ces données dans le chemin, notre cercle sauterait de 100 unités vers la droite et de 102 unités vers le bas, *puis* commencerait à se mouvoir le long du chemin relatif à sa nouvelle position. Donc gardez bien ceci à l'esprit lorsque vous préparerez le chemin de déplacement de vos animations.

Si on les utilise, les attributs `from`, `by`, `to` et `values` spécifient une forme sur le canevas en cours qui représente le chemin de déplacement.

Spécifier le chemin avec l'élément `mpath`

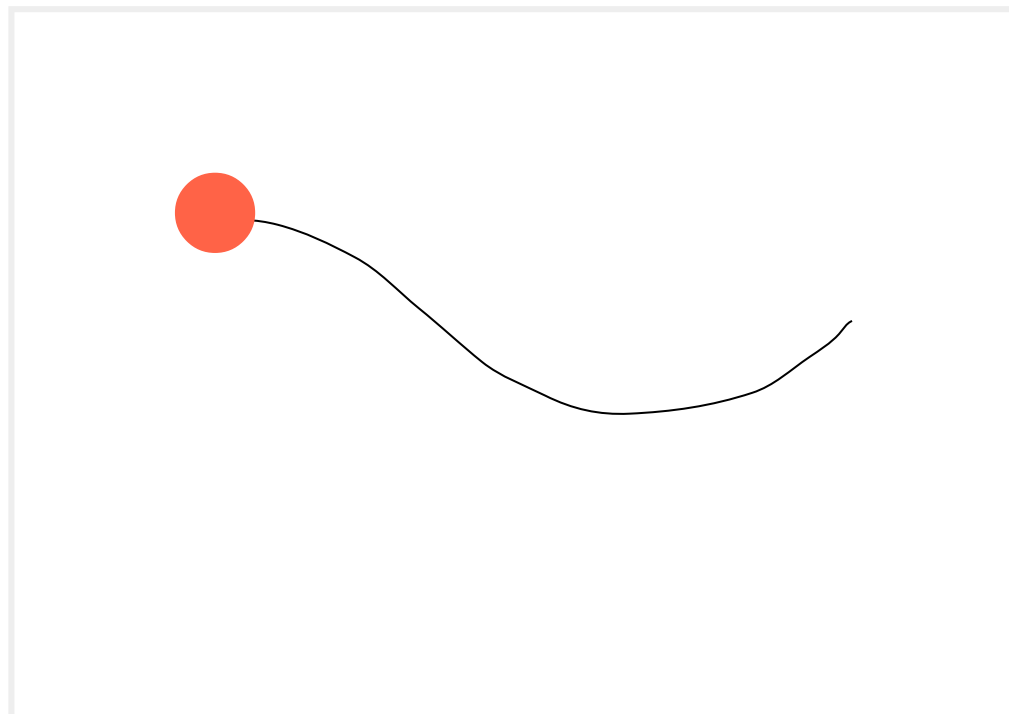
Il existe une autre façon de spécifier un chemin de déplacement. Au lieu d'utiliser l'attribut relatif `path`, on peut référencer un chemin externe grâce à l'élément `<mpath>`. Le `<mpath>`, qui est un enfant de l'élément `<animateMotion>` référence alors le chemin externe avec l'attribut `xlink:href`.

```
//SVG
<animateMotion xlink:href="#circle" dur="1s" begin="click" fill="freeze">
<mpath xlink:href="#motionPath" />
</animateMotion>
```

Le chemin de déplacement `<path>` peut être défini n'importe où dans le document. Il peut même littéralement être défini dans un élément `<defs>` et ne pas être rendu sur le canevas. Dans l'exemple suivant, le chemin est rendu parce que la plupart du temps vous voudrez montrer le chemin que suit l'élément.

Notez que la position du cercle est "multipliée" ou "transformée" par les coordonnées dans les données du chemin.

Dans l'exemple suivant, nous avons un chemin situé au milieu du canevas. Le cercle est positionné au début du chemin. Cependant, lorsque le chemin de déplacement est appliqué, le cercle ne commence pas à se mouvoir depuis sa position courante. Regardez la démo pour une meilleure explication.



Click on the circle to animate it.

Vous avez remarqué la façon dont le cercle suit la même forme que le chemin mais à partir d'une position différente ? Ceci est dû au fait que la position du cercle est transformée par les valeurs des données du chemin.

Pour éviter cela, on peut commencer avec un cercle positionné à (0,0).

Une autre façon de faire est d'appliquer une transformation qui “remet à zéro” les coordonnées du cercle.

L'exemple qui suit est une version modifiée de la démo précédente, utilisant un chemin fermé et répétant l'animation de manière indéfinie.

HTML

CSS

Result

EDIT ON
CODEPEN

Prévalence de règles pour animateMotion

Puisqu'il existe plus d'une façon de faire la même chose avec `animateMotion`, il est logique d'avoir une prévalence de certaines règles sur d'autres :

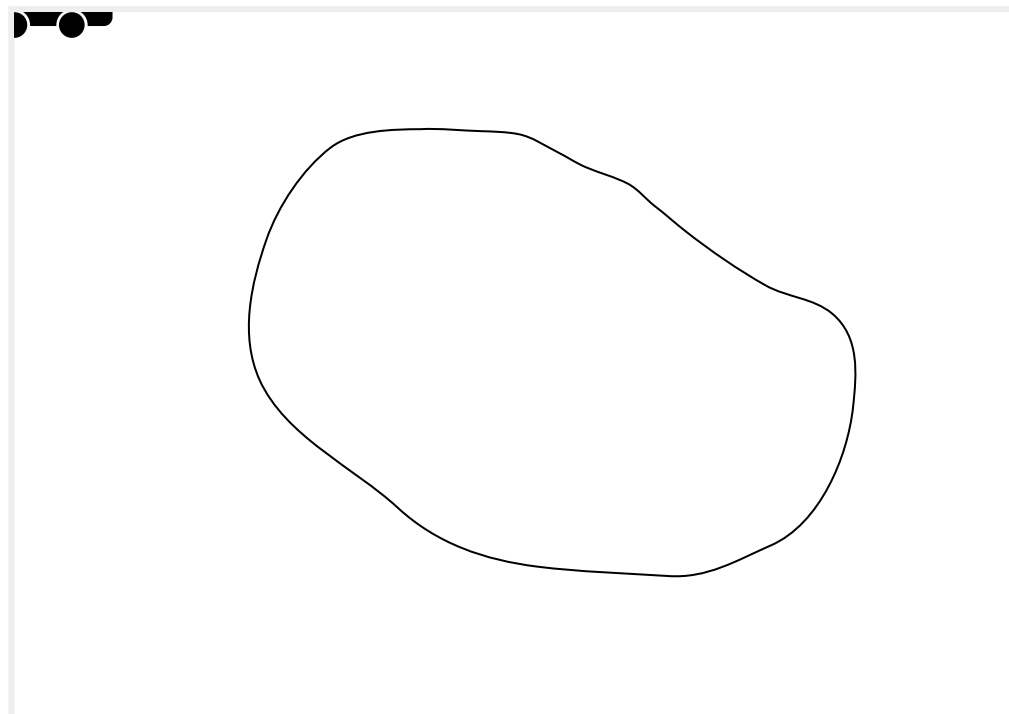
- Concernant la définition du chemin de déplacement, l'élément `mpath` prévaut sur l'attribut `path`, qui lui-même prévaut sur `values`, qui prévaut sur `from`, `by` et `to`.
- Concernant la détermination des points correspondant aux attributs `keyTimes`, l'attribut `keyPoints` prévaut sur `path` qui lui-même prévaut sur `values`, qui prévaut sur `from`, `by` et `to`.

Fixer l'orientation d'un élément le long d'un chemin avec `rotate`

Dans l'exemple précédent, l'élément que nous animions le long du chemin était un cercle. Mais que se passe-t-il si nous animons un élément qui a une certaine orientation, par exemple l'icône d'une voiture ? Nous nous servirons de l'icône [conçue par Freepik](#).

Dans cet exemple, j'ai remplacé le cercle par un groupe ayant un ID de "car", qui contient l'élément constituant le groupe. Puis, afin d'éviter le problème de saut brusque rencontré précédemment, j'ai appliqué une transformation à la voiture qui la translate de façon telle qu'elle se retrouve en position initiale à (0,0). Les valeurs de la transformation correspondent aux coordonnées du point où le premier chemin de la voiture commence à être dessiné (juste après la commande `move M`).

La voiture se déplace le long du chemin mais... voici à quoi ça ressemble :



L'orientation de la voiture est fixe, elle ne change pas pour s'adapter au chemin de déplacement. Pour modifier cela, nous allons utiliser l'attribut `rotate`.

L'attribut `rotate` peut prendre l'une de ces trois valeurs :

- `auto` : indique que l'objet pivote dans le temps en fonction de l'angle de la direction (c'est à dire le vecteur directionnel tangent) du chemin de déplacement.
- `auto-reverse` : indique que l'objet pivote selon l'angle de direction + 180 degrés.
- un nombre : indique que l'élément se voit appliquer une transformation constante, où l'angle de rotation est le nombre de degrés spécifié.

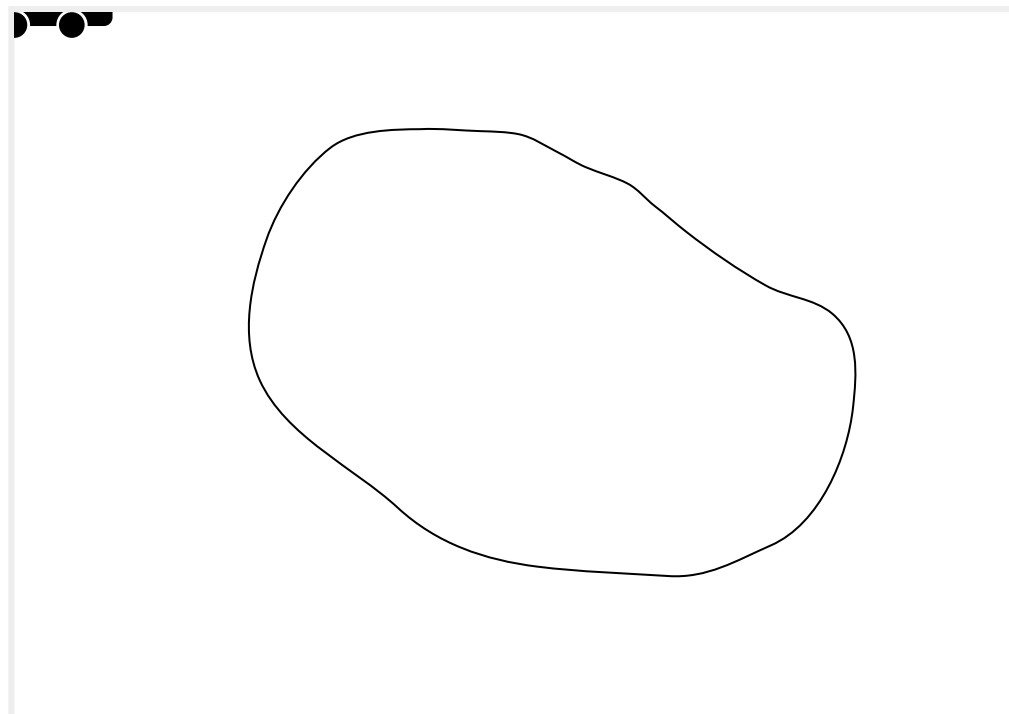
Pour corriger l'orientation de la voiture, nous allons régler la valeur de rotation sur `auto`. Nous obtenons le résultat suivant :

HTML

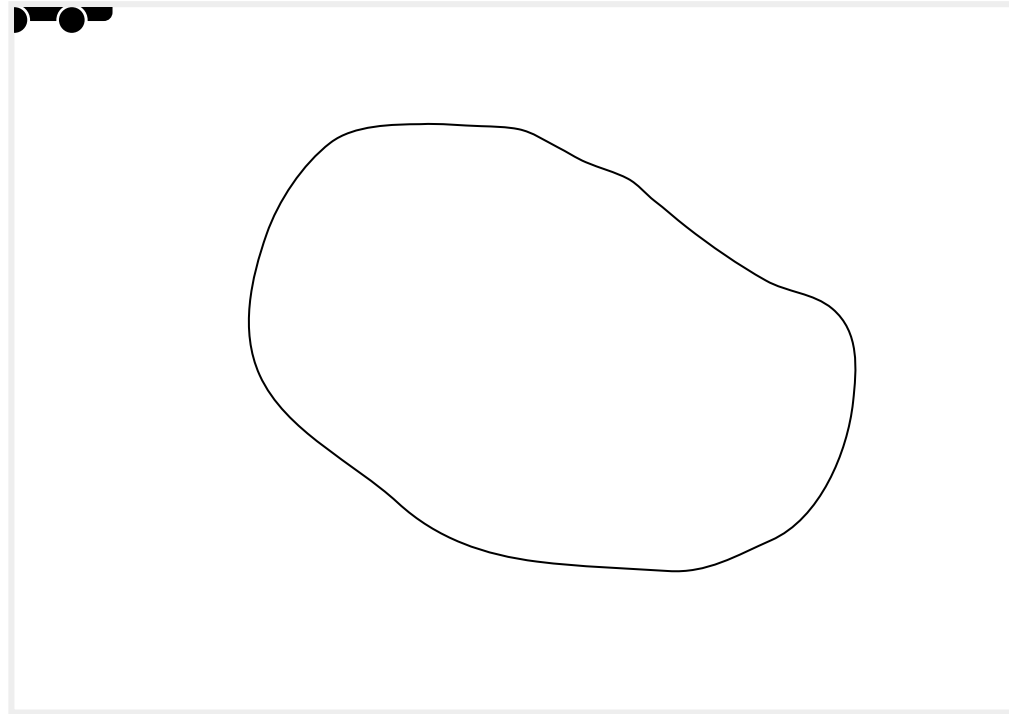
CSS

Result

EDIT ON
CODEPEN



Si vous préférez que la voiture se retrouve en dehors du chemin plutôt qu'à l'intérieur, `auto-reverse` vous permet d'y parvenir :

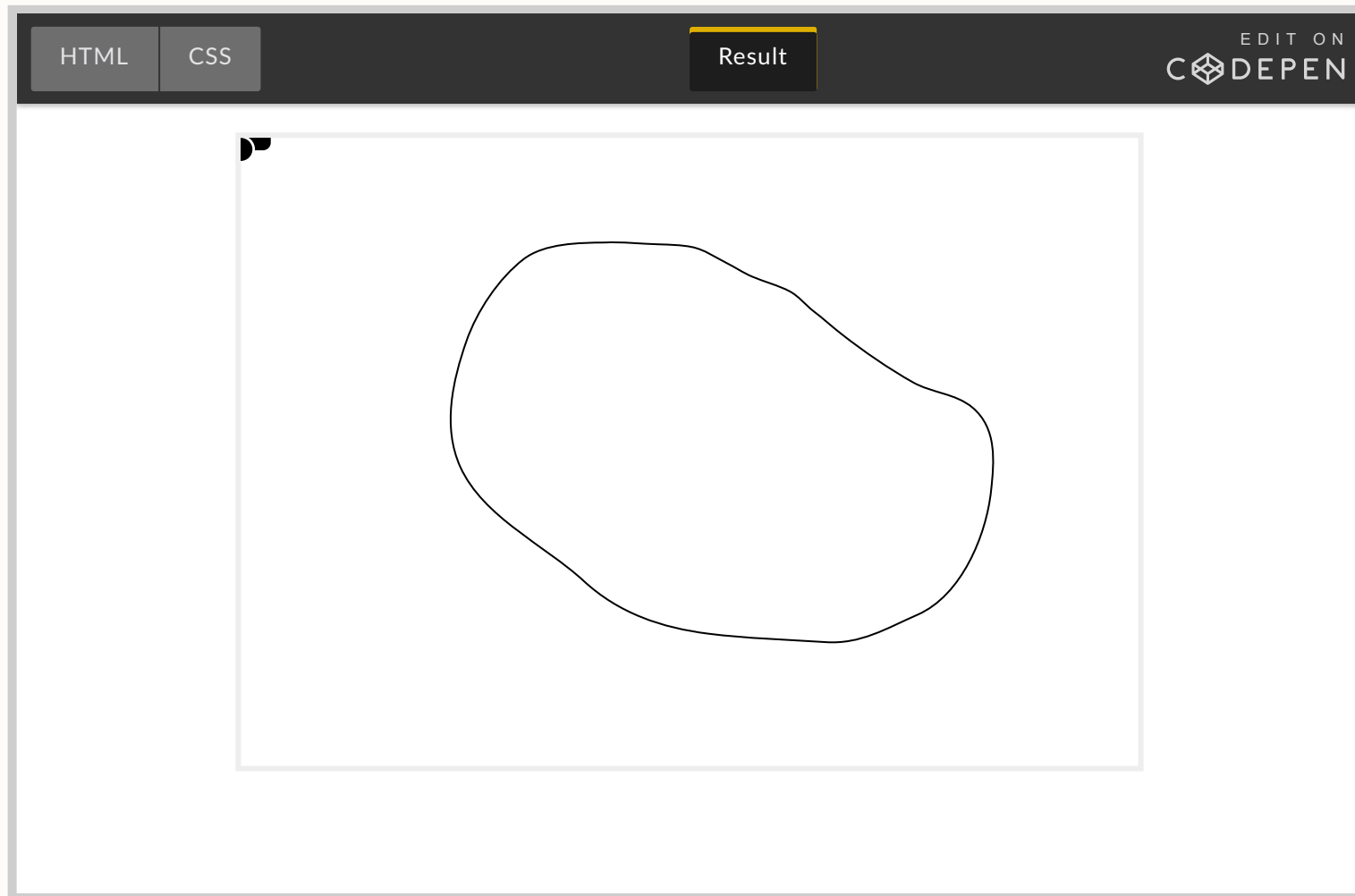


C'est mieux, mais nous avons encore un petit problème : la voiture a l'air d'avancer à l'envers le long du chemin, comme si elle faisait de la marche arrière ! Pour corriger cela, nous devons faire pivoter la voiture sur son axe des y. On y parvient avec `scale`. Donc si nous appliquons la transformation au groupe `g` ayant l'ID de `car`, la voiture avancera comme prévu. La transformation `scale` est enchaînée avec les autres translations :

```
//SVG
```

```
<g id="car" transform="scale (-1, 1) translate(-234.4, -182.8)">
```

Et voici la démo finale :



Contrôler la distance d'animation le long du chemin

L'attribut `keyPoints` nous offre la possibilité de spécifier la progression le long du chemin de déplacement pour chacune des valeurs de `keyTimes`. Si on utilise des `keyPoints`, `keyTimes` prendra les valeurs de `keyPoints` au lieu de celles spécifiées dans une liste de `values`.

`keyPoints` accepte une liste de valeurs décimales entre 0 et 1, séparées par des points-virgules, et il indique à quel endroit l'objet doit se déplacer à un moment donné, spécifié par les valeurs `keyTimes` correspondantes. Les calculs de distances sont déterminés par les algorithmes du navigateur. Chaque valeur de progression dans la liste correspond à une valeur dans la liste de l'attribut `keyTimes`. Si une liste de `keyPoints` est spécifiée, il doit y avoir exactement autant de valeurs dans la liste `keyPoints` que dans la liste `keyTimes`.

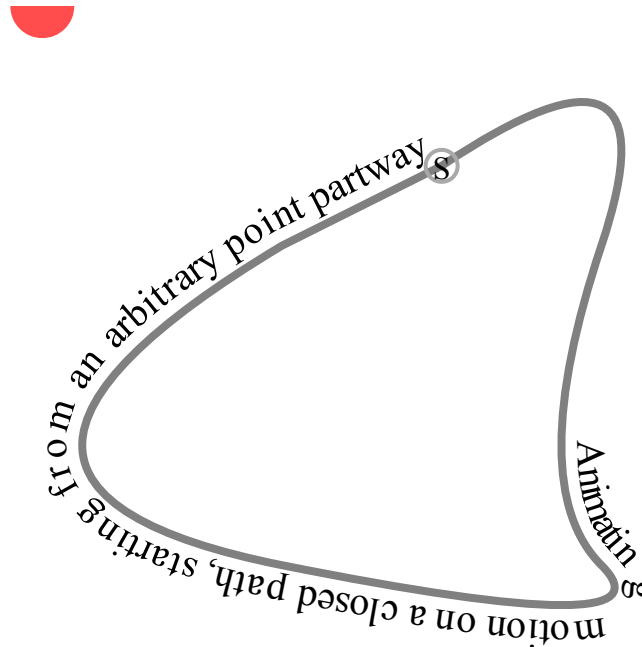
Une chose importante à noter ici est qu'il faut régler la valeur de `calcMode` sur `linear` pour que `keyPoints` fonctionne.

L'exemple qui suit est d'Amelia Bellamy-Royds (dont vous devriez absolument visiter le [profil CodePen](#)) il utilise `keyPoints` pour faire comme si on commençait un mouvement depuis un point en retrait, car nous n'avons pas cette possibilité par défaut actuellement dans SMIL.

There is no pre-defined way in SVG1.1 to define an offset for `<animateMotion>`. However, you can mimic this behaviour by using key points and times, jumping from the end to the beginning of the path instantaneously.

The math to calculate keyPoints and keyTimes for the forward movement is as follows:

```
keyPoints
  (startPosition)
  1
  0
  (startPosition)
keyTimes
  0
  (1 - startPosition)
  (1 - startPosition)
  1
```




Déplacer un texte le long d'un chemin arbitraire

Le déplacement d'un texte le long d'un chemin est différent du déplacement des autres éléments SVG le long d'un chemin. Pour animer du texte, vous devrez utiliser l'élément `<animate>` et non pas l'élément `<animateMotion>`.

Tout d'abord, positionnons le texte le long d'un chemin. On peut le faire en emboîtant un élément `<textPath>` à l'intérieur de l'élément `<text>`. Le texte qui sera positionné le long d'un chemin sera défini à l'intérieur de l'élément `<textPath>` et non comme un enfant de l'élément `<text>`.

Le `textPath` va référencer le chemin que nous voulons utiliser, comme nous l'avons fait dans les exemples précédents. Le chemin référencé peut soit être rendu sur le canvas, soit défini à l'intérieur de `<defs>`. Regardez le code de la démo suivante en cliquant sur [html](#) :

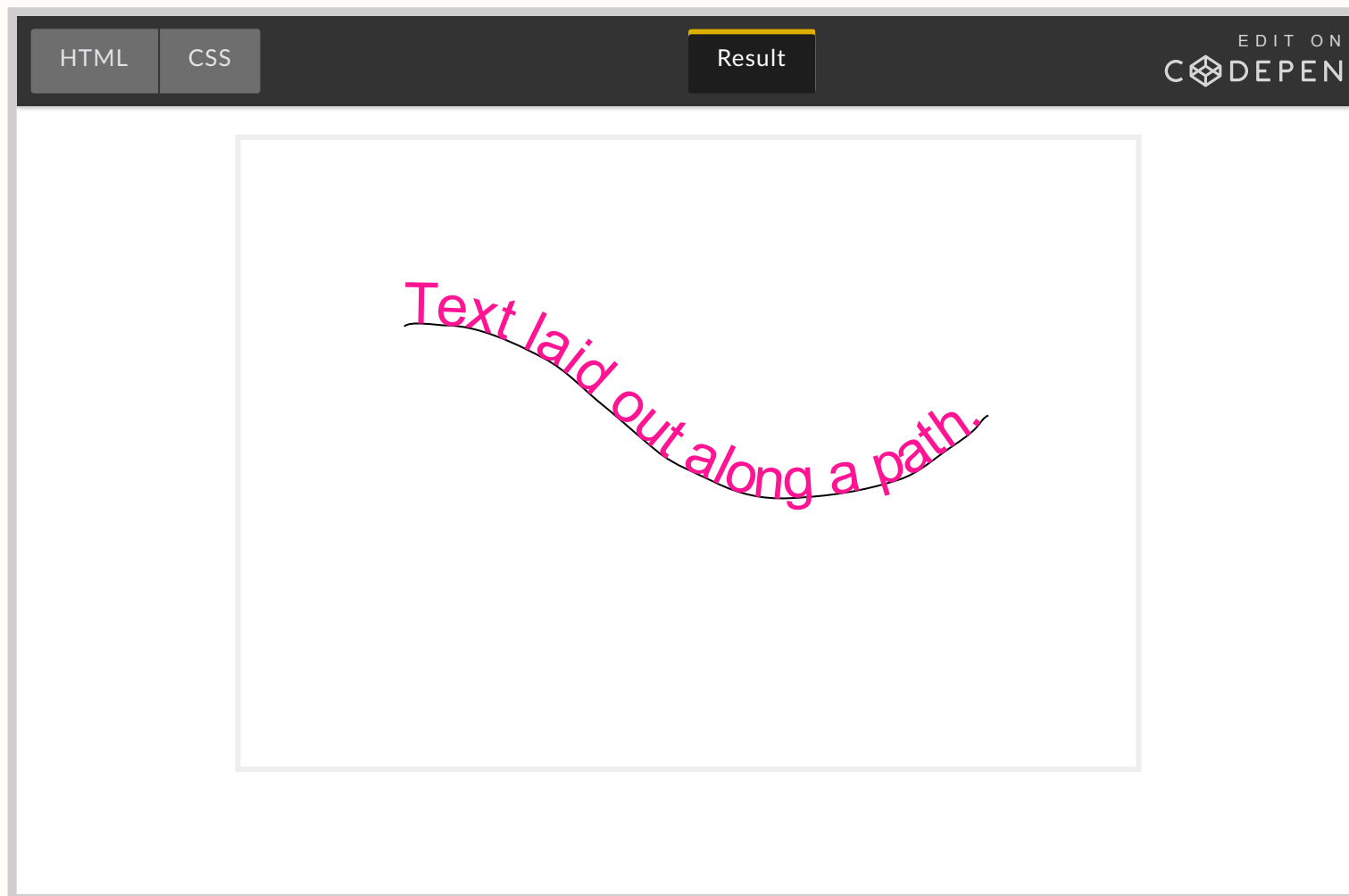


Text laid out along a path.

Pour animer le texte le long de ce chemin, nous allons utiliser l'élément `<animate>` pour animer l'attribut `startOffset`.

Le `startOffset` représente le décalage éventuel du texte sur le chemin. 0% représente le début du chemin, 100% la fin, donc si l'offset est réglé à 50% le texte commencera à mi-chemin.

En animant le `startOffset`, nous allons créer l'effet du texte qui se déplace sur le chemin. Regardez le code pour plus de détails.



Animer les transformations

L'élément `<animateTransform>` anime un attribut de transformation sur un élément cible, permettant par là-même aux animations de contrôler la translation, le redimensionnement, la rotation et la déformation. Il prend les mêmes attributs que ceux mentionnés pour l'élément `<animate>`, plus un : `type`.

L'attribut `type` est utilisé pour spécifier le type de transformation animé. Il prend l'une de ces cinq valeurs : `translate`, `scale`, `rotate`, `skewX` et `skewY`.

Les attributs `from`, `by` et `to` prennent une valeur exprimée avec la même syntaxe que celle qui est disponible pour le type de transformation concerné :

- Pour un `type="translate"` chaque valeur individuelle est exprimée comme `<tx> [,<ty>]`.
- Pour un `type="scale"` chaque valeur individuelle est exprimée comme `<sx> [,<sy>]`.
- Pour un `type="rotate"` chaque valeur individuelle est exprimée comme `<rotate-angle> [<cx> <cy>]`.
- Pour un `type="skewX"` et `type="skewY"` chaque valeur individuelle est exprimée comme `<skew-angle>`.

Si vous n'êtes pas habitué à la syntaxe des fonctions de l'attribut `transform` de SVG, je vous recommande de lire l'article que j'ai écrit à ce sujet : [Understanding SVG Coordinate Systems and](#)

Transformations (Part 2): The Transform Attribute, avant de continuer avec ce guide.

Revenons à une démo précédente, celle où nous faisons tourner sur lui-même le rectangle rose en utilisant l'élément `<animateTransform>`. Le code de la rotation ressemble à ceci :

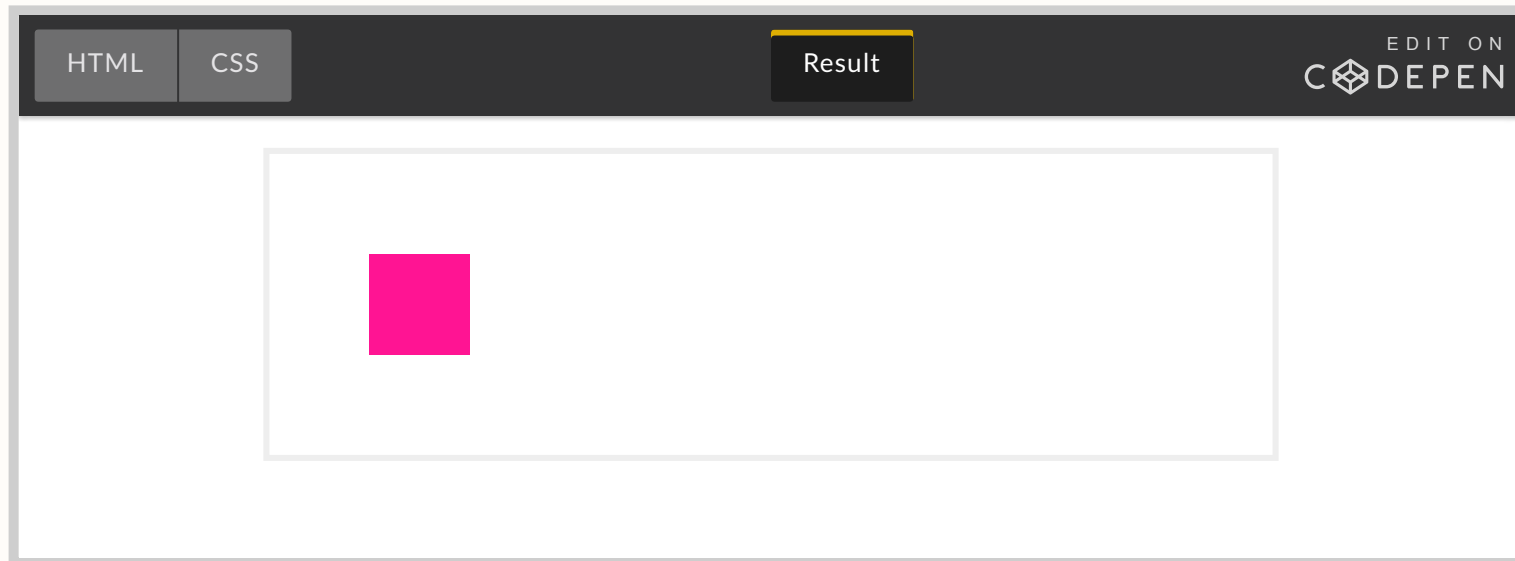
```
//SVG
<rect id="deepPink-rectangle" width="50" height="50" x="50" y="50" fill="deepPink" />

<animateTransform
  xlink:href="#deepPink-rectangle"
  attributeName="transform"
  attributeType="XML"
  type="rotate"
  from="0 75 75"
  to="360 75 75"
  dur="2s"
  begin="0s"
  repeatCount="indefinite"
  fill="freeze"
/>
```

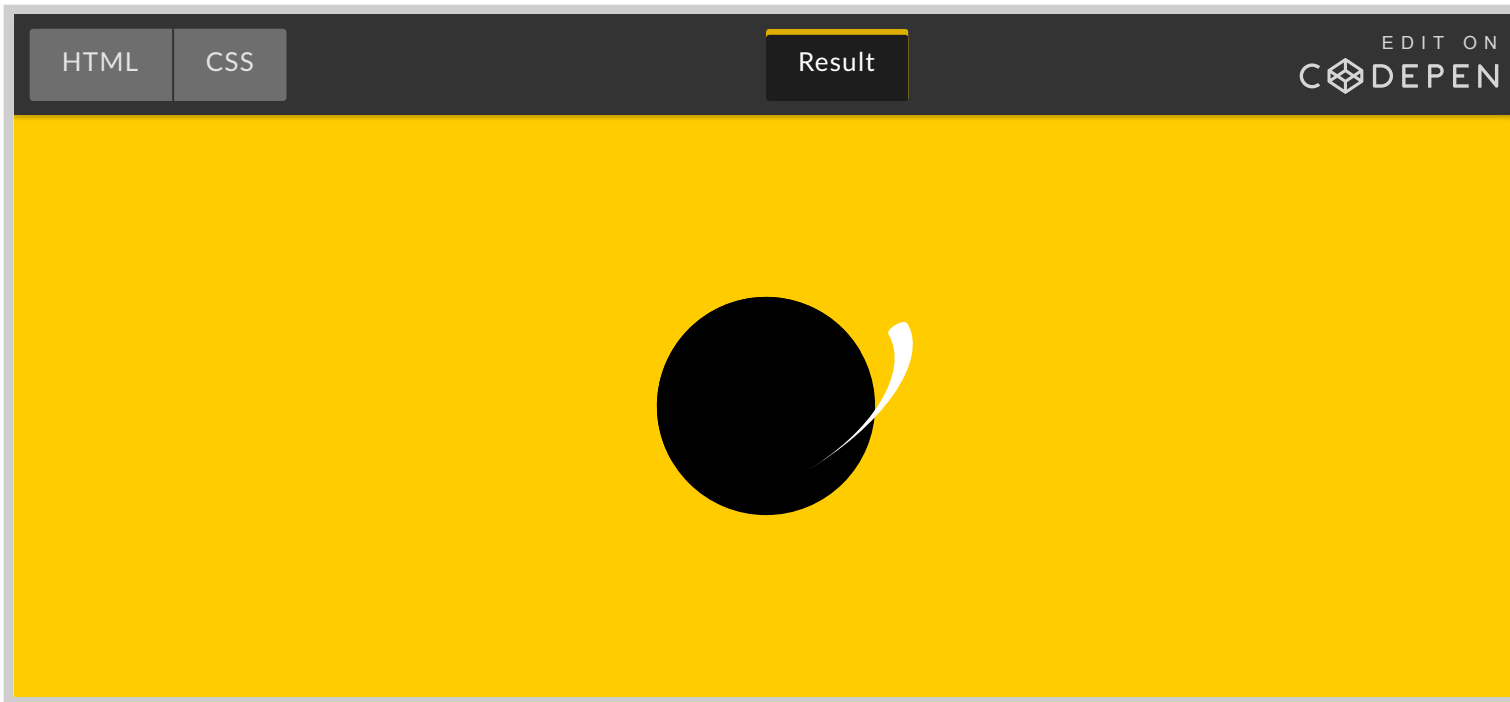
Les attributs `from` et `to` spécifient l'angle de rotation (début et fin) et le centre de la rotation (75, 75). Dans les deux, le centre de la rotation reste inchangé bien sûr. Si l'on ne spécifie pas le centre, il

sera par défaut le coin supérieur gauche du canevas SVG.

La démo live du code est la suivante :



Et voici un exemple amusant par Gabriel, avec un simple `animateTransform` :



Animer une transformation unique est simple, toutefois les choses peuvent devenir compliquées lorsqu'on inclut des transformations multiples, en particulier du fait qu'une `animateTransform` peut en écraser une autre, ce qui fait qu'au lieu d'ajouter et d'enchaîner des effets, vous terminerez avec l'inverse. Ajoutez à cela la manière dont fonctionnent les systèmes de coordonnées SVG (cf. mon article cité plus haut). Pour transformer les SVG, je vous recommande d'utiliser les transformations CSS.

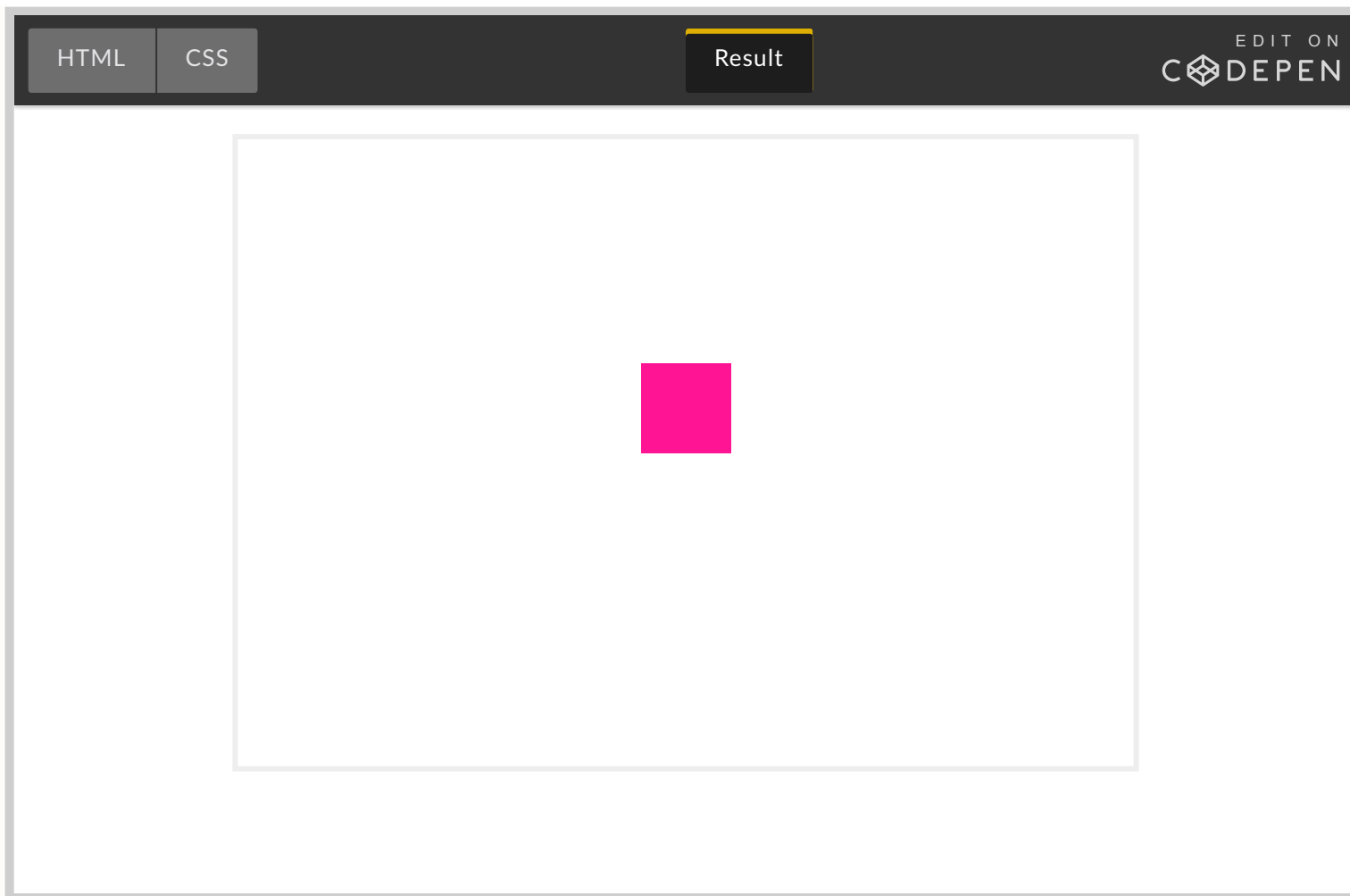
L'élément `set`

L'élément `<set>` offre une manière simple de régler la valeur d'un attribut pour une durée donnée. Il accepte tous les types d'attributs, dont ceux qui ne peuvent pas être raisonnablement interpolés, tels que les chaînes de caractères et les valeurs booléennes. L'élément `set` n'est pas additif. Les attributs additifs et accumulatifs ne sont pas permis et seront ignorés si on les utilise.

Puisque `<set>` est utilisé pour donner une valeur spécifique à un élément, pour une durée déterminée, il n'accepte pas tous les attributs mentionnés pour les éléments d'animation précédents. Par exemple, il n'a pas d'attribut `from` ou `by` car les valeurs ne changent pas progressivement sur la période de temps.

Pour `set`, vous pouvez spécifier l'élément que vous ciblez, le nom et le type d'attribut, la valeur `to`, et le timing d'animation peut être contrôlé via `begin`, `dur`, `end`, `min`, `max`, `restart`, `repeatCount`, `repeatDur` et `fill`.

Dans l'exemple qui suit, on fixe (`set`) la couleur du rectangle tournant au bleu lorsqu'on le clique. La couleur reste bleue pendant 3 secondes, puis revient à sa valeur d'origine. À chaque fois qu'on clique sur le rectangle, l'animation `set` est déclenchée et la couleur change pour 3 secondes.



Éléments, attributs et propriétés peuvent être animés

Tous les attributs SVG ne peuvent pas être animés, et parmi ceux qui peuvent l'être, tous ne peuvent utiliser l'intégralité des éléments d'animation. Pour une liste complète des attributs animables, et un

tableau montrant lesquels peuvent être animés par quels éléments, vous pouvez vous référer à [cette section de la spécification SVG Animation](#).

Pour conclure

SMIL a un potentiel énorme, je n'ai fait qu'effleurer la surface et je n'ai abordé que les bases les plus simples. De nombreux effets très impressionnants peuvent être réalisés, en particulier ceux qui impliquent le morphing et la transformation des formes. *The sky is the limit*. Soyez fous ! et n'oubliez pas de partager ce que vous créez avec la communauté ! Merci de votre lecture.

Intéressé par SVG ? Retrouvez une liste des meilleurs [articles](#) et [ressources](#) du web.

📖 [Tous les articles sur SVG](#) parus dans la Cascade.

Ressources complémentaires

Un bel exemple d'utilisation de SMIL, dans le jeu en ligne [esviji](#) de [Nicolas Hoizey](#).

(publicité)

Article original paru le 13 octobre 2014 dans [CSS-Tricks](#)

Traduit avec l'aimable autorisation de CSS-Tricks et de l'auteur.

Copyright CSS-Tricks © 2014.

Sur l'auteur : Sara Soueidan est une intégratrice web libanaise. Elle aime enseigner et écrit des tutoriels sur son [blog](#) et sur [Codrops](#), dont elle est un des membres actifs. Vous pouvez la suivre sur [Twitter](#) et sur [Github](#).



pierre choffé

Manager d'ensembles musicaux, Traducteur, en route vers la Planète SemanticWeb à bord du vaisseau FRBRoo, pour la description, le partage et l'enrichissement des données musicales.



Paris, France

Share this post



READ THIS NEXT

Masquer et détourer en CSS

YOU MIGHT ENJOY

CSS animation, une introduction

