Chapitre 6 : Systèmes de coordonnées et transformations

Système de coordonnées: nous ne pouvons nous en passer

Nous allons beaucoup parler de coordonnées dans ce chapitre. Qu'en savons nous?
Le document SVG s'ouvre avec un système de coordonnées par défaut – le système de coordonnées utilisateur.

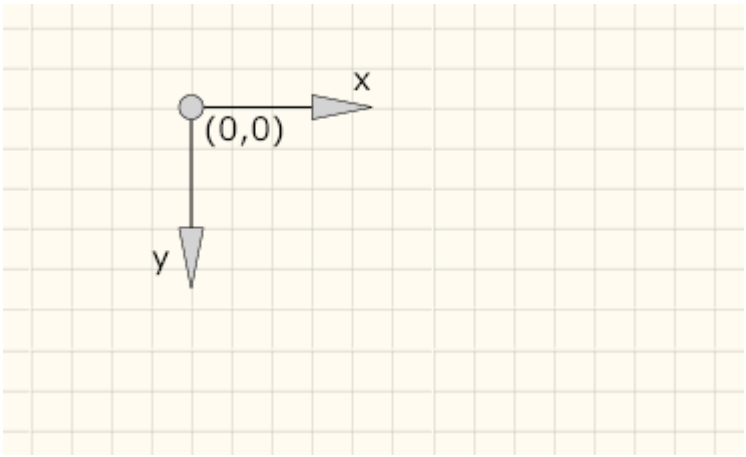

Figure 6-1. Système de coordonnées utilisateur

Avec ce système, nous définissons un espace qui est théoriquement infini dans ses deux dimensions. Pour repérer un point, nous devons avoir un repère et une unité. SVG nous fournit une unité, l'unité utilisateur qui est initialement égale au pixel. Ce pixel est le plus petit point visible sur un écran. Les coordonnées et la position, la taille des objets dépendent donc de l'écran ou l'imprimante qui rendra ce document

L'espace est infini, cependant votre écran ne l'est pas, aussi les spécifications SVG définissent une région rectangulaire finie dans laquelle seront rendus les objets graphiques. Cette fenêtre est nommée '**Viewport**', nous traduirons ce terme par fenêtre de visualisation .

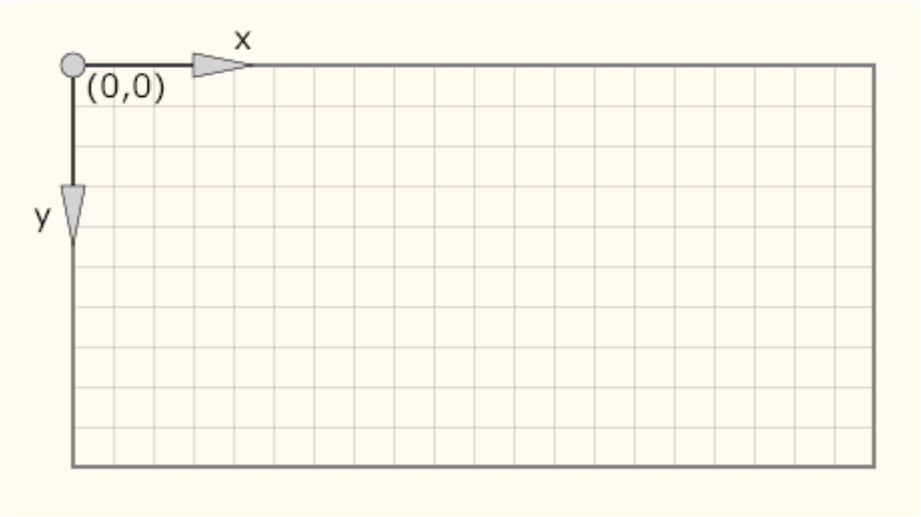

Figure 6-2. 'ViewPort' ou fenêtre de visualisation

```
<svg width="200" height="100">
```

Vous pouvez définir les dimensions de cette fenêtre avec les attributs 'width' et 'height' dans l'élément racine <svg>. Le quadrillage dans notre figure a un espacement de 10 unités. L'angle supérieur gauche coïncide avec l'origine du système de coordonnées, l'axe des x

horizontal est dirigée vers la droite et l'axe des y vertical vers le bas. Ceci dit, nous pouvons prévoir exactement la taille et la position de nos objets graphiques. Notons simplement que l'origine de ce système de coordonnées se déplacera si nous faisons un panoramique ou un zoom sur l'image.

Les mathématiciens et ingénieurs remarqueront que l'orientation de ce repère est inhabituel, car pour eux l'axe des y devrait se diriger vers le haut. Ceci aura aussi des conséquences pour l'orientation des angles, le sens positif sera le sens des aiguilles d'une montre comme nous l'avons vu pour les arcs d'ellipse au chapitre 4. Cependant, ce type de repère est courant dans le champ du graphisme par ordinateur.

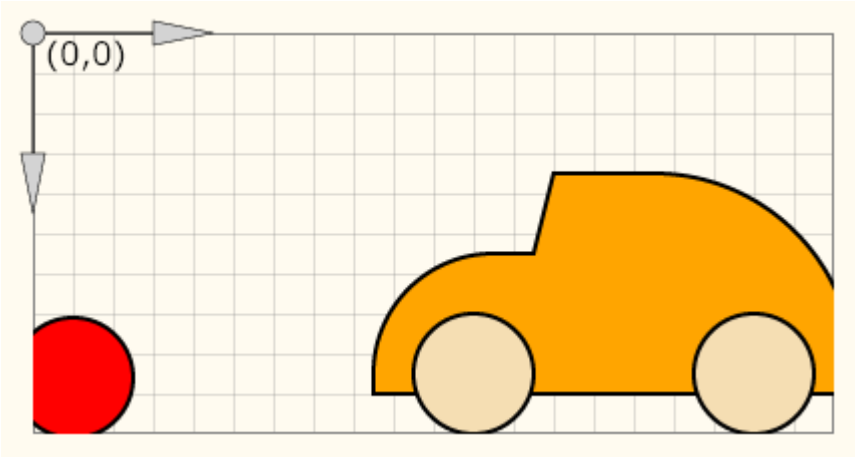

Figure 6-3. Quelques objets

```
<svg width=”200” height=”100”>
  <circle cx="10" cy="86" r="15" fill="red" stroke="black" />
  <!-- code pour l'automobile -->
</svg>
```

Pour ces objets graphiques, seule la partie intérieure à la fenêtre de visualisation est rendue. Les objets qui sont totalement à l'extérieur sont invisibles.

Repères multiples

SVG permet de définir plusieurs fenêtres de visualisation comme descendants de l'élément racine.

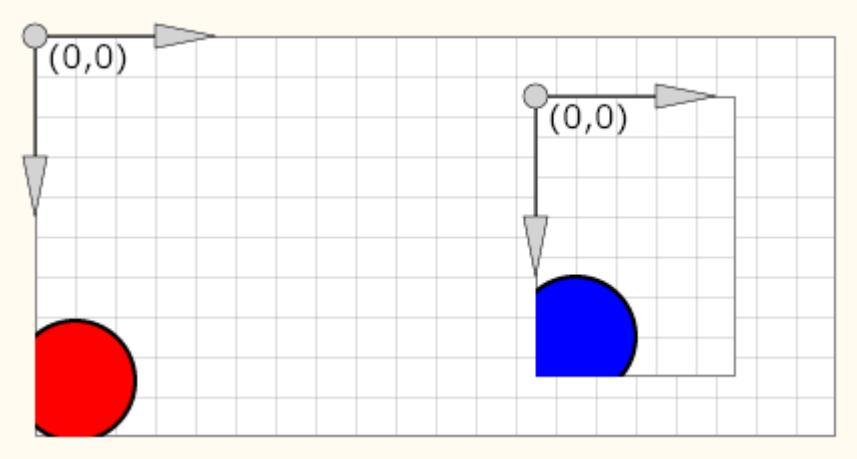

Figure 6-4. Cercles dans différents repères

```
<svg width=”200” height=”100”>
  <circle cx="10" cy="86" r="15" fill="red" stroke="black"/>
  <svg x="125" y="15" width="50" height="70">
```

```
      <circle cx="10" cy="60" r="15" fill="blue" stroke="black"/>
    </svg>
  </svg>
```

Dans cet exemple le second repère est défini à l'intérieur du repère principal, son origine est localisée en (125,15), il a 50 unités de large et 70 unités de haut, tout ceci exprimé dans le repère parent, ici le repère de l'élément racine.

La position de ce repère avec les attributs 'x' et 'y' est possible pour tous les éléments <svg> excepté l'élément racine. Ce nouveau repère établit son propre système de coordonnées avec l'origine au coin supérieur gauche et l'unité utilisateur par défaut. Tous les objets descendants de cet élément <svg> utiliseront ce système de coordonnées. La représentation de ces objets sera limité à l'intérieur de la fenêtre de visualisation si vous ne précisez pas l'attribut 'overflow' avec overflow=”visible”.

Le dessin montre que ces deux repères utilisent la même unité, ainsi deux cercles de même rayon auront la même taille dans les deux repères. Pouvons nous changer cette unité? Oui en utilisant l'attribut 'viewBox'.

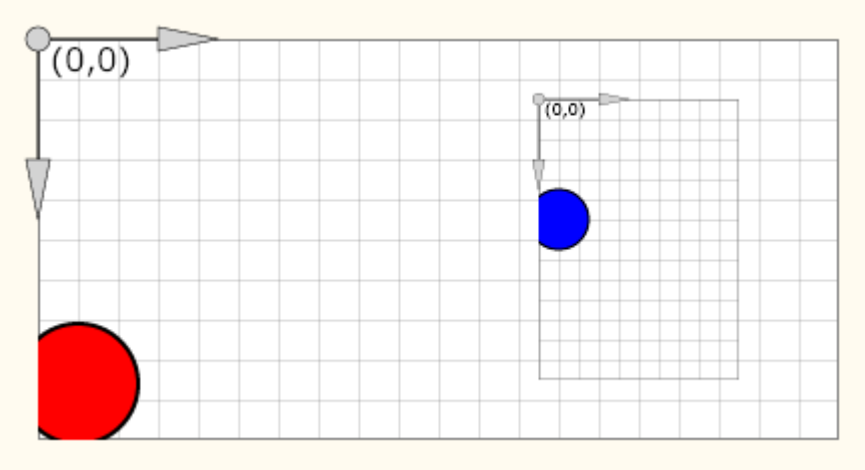

Figure 6-5. Changement d'unité

```
<svg width=”200” height=”100”>
  <circle cx="10" cy="86" r="15" fill="red" stroke="black" />
  <svg x="135" y="15" width="50" height="70" viewBox="0 0 100 140">
     <circle cx="10" cy="60" r="15" fill="blue" stroke="black" />
  </svg>
</svg>
```

L'attribut 'viewBox' est défini par quatre nombres xmin, ymin, width, height. Avec ceux-ci vous pouvez spécifier la position du repère et les unités.
Attention, ces quatre valeurs sont exprimées dans le repère de l'élément <svg> et non dans le repère de son ascendant, ce qui n'est pas le cas des attributs 'x', 'y', 'width' et 'height'.

Sur notre exemple, l'origine restera au coin supérieur gauche car xmin=0 et ymin=0, mais nous avons une largeur de 100 pour une fenêtre de 50 unités, nous aurons donc comme unité la moitié de l'unité utilisée par l'élément <svg> ascendant. Si la hauteur et la largeur du repère ne sont pas proportionnels aux dimensions de la fenêtre, nous verrons plus loin le problème posé.

Syntaxe:

```
viewBox=”xmin, ymin, width, height”
```

`xmin`	x du coin supérieur gauche
`ymin`	y du coin supérieur gauche
`width`	largeur de la fenêtre

`height` hauteur de la fenêtre

Système local de coordonnées

Quand nous avons vu les éléments <g> et <use> au chapitre 3, nous n'avons pas traité le choix du repère. Nous allons le faire maintenant. Nous reprenons notre exemple de rack et nous définissons une palette constituée de quatre rectangles.

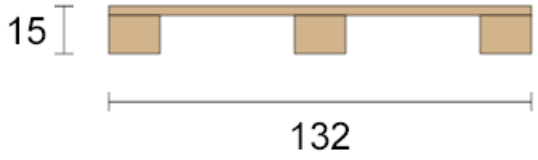

Figure 6-6. Une palette pour le rack

Cet objet va être réutilisé, aussi nous le définissons comme un groupe dans la section <defs>. Comment choisir les coordonnées pour les éléments <rect>. Comme les dimensions de cette palette sont définies, nous n'avons qu'à choisir une origine pour le repère, prenons le coin supérieur gauche comme origine du repère, il aura les coordonnées (0,0). Nous définissons donc le repère représenté sur la figure 6-7.

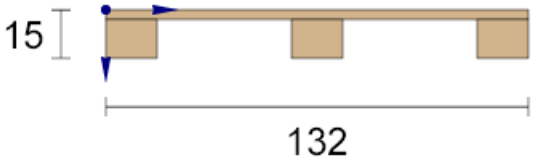

Figure 6-7. La palette et son repère

```
<g id="palette" stroke="black" fill="tan" >
    <defs>
      <rect id="bloc" width="16" height="12" />
    </defs>
    <rect x="0" y="0" width="132" height="3" />
    <use xlink:href="#bloc" x="0" y="3" />
    <use xlink:href="#bloc" x="58" y="3" />
    <use xlink:href="#bloc" x="116" y="3" />
  </g>
```

La planche supérieure a une épaisseur de 3, les chevrons auront une largeur de 16 et une hauteur de 12. Nous pouvons ainsi calculer les coordonnées de tous ces éléments. Comme tout ceci est en bois, nous choisissons la couleur de remplissage 'tan' pour tous ces rectangles.

Vous devez vous demander: “Où est défini le système de coordonnées dans ce code? ”. En fait, nulle part, tout au moins pas de manière explicite. Quand nous donnons des valeurs aux attributs, x="58" et y="3", nous définissons implicitement un système de coordonnées. Nous pouvons imaginer le système de coordonnées lié à cette palette, nous le nommerons sytème local de coordonnées du groupe. Nous pouvons aussi nommer système de coordonnées de référence le système de coordonnées du container (le parent du groupe en général) ou de la fenêtre parente.
Nous pouvons maintenant nous demander si ce choix arbitraire du système local de la palette est judicieux. Ces palettes doivent être placées dans le rack.

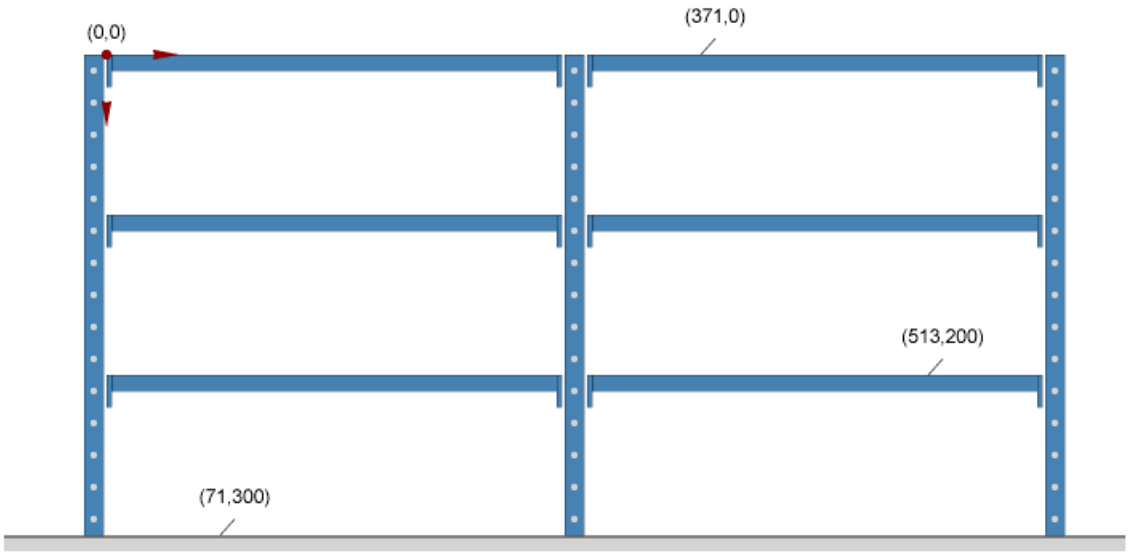

Figure 6-8. Le rack pour les palettes

Reprenons le rack du chapitre 3, nous voulons y placer trois palettes aux points indiqués sur la figure. Ce rack a lui aussi un système local de coordonnées représenté sur la figure. Les positions où nous voulons placer des palettes sont exprimées dans ce repère. Nous prenons donc la palette que nous avons définie.

```
<use xlink:href="#palette" x="71" y="300" />
```

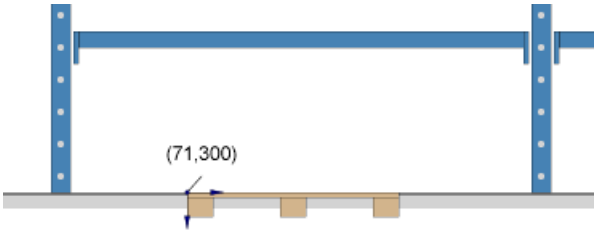

Figure 6-9. Mettre en place une palette …

Bof! Ce n'est pas très réussi, mais avec un peu de calcul …

```
x_palette = 71 - largeur_palette/2
y_palette = 300 - hauteur_palette
```

Nous y arrivons.

```
<use xlink:href="#palette" x="5" y="285" />
```

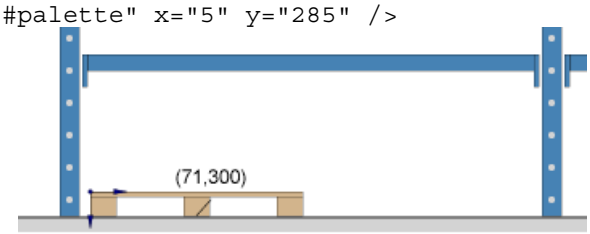

Figure 6-10. C'est mieux!

Mais, nous pouvons supposer que les palettes n'ont pas toutes la même taille et qu'il va falloir calculer chaque fois que nous plaçons une palette. Pour éviter ces calculs, nous pouvons changer le repère local de la palette et placer l'origine du repère au milieu du bas du chevron central. Ce qui nous donne:

```
<g id="palette" stroke="black" fill="tan" >
  <defs>
```

```
      <rect id="bloc" width="16" height="12" />
    </defs>
    <rect x="-66"   y="-15" width="132" height="3" />
    <use xlink:href="#bloc" x="-66" y="-12" />
    <use xlink:href="#bloc" x="-8" y="-12" />
    <use xlink:href="#bloc" x="50" y="-12" />
  </g>
```

Avec ce repère pour la palette, plus de calcul pour la positionner correctement.

```
<use xlink:href="#palette" x="71" y="300" />
```

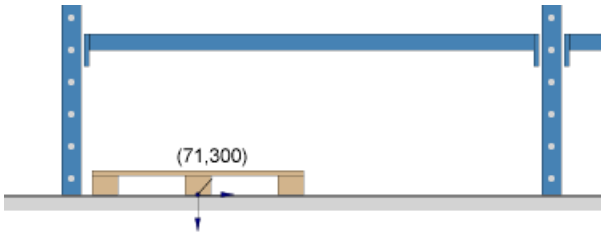

Figure 6-11. Un repère pour la palette

Grâce à cet exemple, nous réalisons que le choix de l'origine de notre système local de coordonnées est très important pour réutiliser ce groupe. Il est maintenant très facile de placer les deux autres palettes.
Nous garnissons même ces palettes.

```
  <use xlink:href="#palette" x="71"  y="300" />
  <use xlink:href="#palette" x="371" y="0"   />
  <use xlink:href="#palette" x="513" y="200" />
```

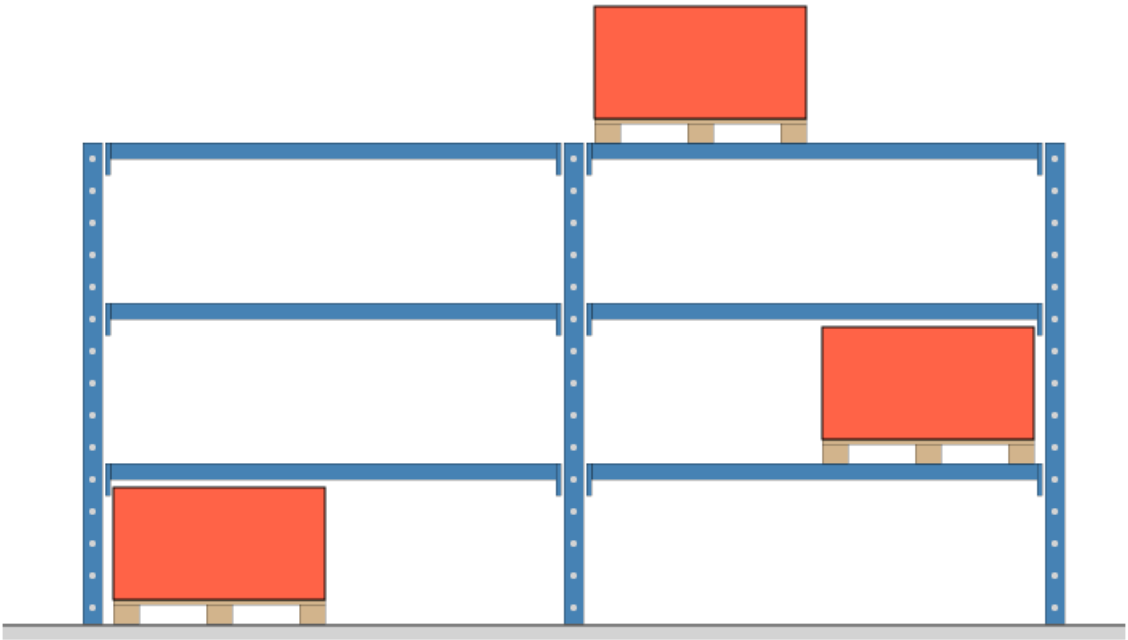

Figure 6-12. Quelques palettes sur le rack

Voici le code complet du document final:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
"http://www.w3.org/TR/SVG/DTD/svg10.dtd">
<svg width="100%" height="100%" xmlns="http://www.w3.org/2000/svg">
  <defs>
    <pattern id="trous" height="20" patternUnits="userSpaceOnUse">
      <rect width="12" height="20" stroke="none" fill="steelblue" />
      <circle cx="6" cy="10" r="2" stroke="none" fill="lightgray" />
```

```
    </pattern>
    <g id="palette" stroke="black" stroke-width="0.2" fill="tan" >
      <defs>
        <rect id="bloc" width="16" height="12" />
      </defs>
      <rect x="-66"   y="-15" width="132" height="3" />
      <use xlink:href="#bloc" x="-66" y="-12" />
      <use xlink:href="#bloc" x="-8" y="-12" />
      <use xlink:href="#bloc" x="50" y="-12" />
    </g>
    <g id="chargement">
       <use xlink:href="#palette" />
       <rect  x="-66"  y="-85"  width="132"  height="70"  stroke="black"  stroke-
width="1" />
    </g>
    <g id="montant" stroke="black" stroke-width="0.2" >
       <rect width="12" height="300" fill="url(#trous)" />
    </g>
    <g id="support" stroke="black" stroke-width="0.2" fill="steelblue" >
       <rect x="3" y="0" width="278" height="10" />
       <rect x="0" y="0" width="3"   height="20" />
       <rect x="281" y="0" width="3" height="20" />
    </g>
    <g id="colonne">
      <use xlink:href="#montant" />
      <use xlink:href="#support" x="14" y="0" />
      <use xlink:href="#support" x="14" y="100" />
      <use xlink:href="#support" x="14" y="200" />
    </g>
    <g id="rack">
      <use xlink:href="#colonne" x="0" y="0" />
      <use xlink:href="#colonne" x="300" y="0" />
      <use xlink:href="#montant" x="600" y="0" />
      <line x1="-50" y1="301" x2="650" y2="301" stroke="black" fill="none" />
      <rect    x="-50"    y="302"    width="700"    height="8"    stroke="none"
fill="lightgray" />
    </g>
  </defs>
  <g transform="translate(100,150)" >
    <use xlink:href="#rack" x="-14" y="0" />
    <use xlink:href="#chargement " x="71" y="300" fill="tomato" />
    <use xlink:href="#chargement " x="371" y="0" fill="tomato" />
    <use xlink:href="#chargement " x="513" y="200" fill="tomato" />
  </g>
</svg>
```

Transformations élémentaires

Nous avons vu comment déplacer un élément <use> en utilisant ses attributs 'x' et 'y' et comment choisir un repère pour un groupe.

Mais si nous voulons faire tourner, changer les dimensions d'un élément? Pour cette tâche, les spécifications SVG fournissent un attribut très puissant – **transform**. Cet attribut peut être appliqué à la grande majorité des éléments graphiques. Il peut être également appliqué à l'élément <g>.

Voici sa syntaxe:

```
   transform = "translate(tx [ ty])
                rotate(angle [cx cy])
                scale(sx [sy])
                skewX(angle)
                skewY(angle)"
```

Nous allons discuter les valeurs possibles de cet attribut séparèment. Nous allons illustrer les transformations avec un objet un peu plus complexe, un élément <use> faisant référence à un groupe. Mais n'oublions pas que l'attribut 'transform' s'applique aussi aux autres objets graphiques.

Nous allons utiliser cette vis en définissant sa géométrie dans son système local de coordonnées représenté en orange sur la figure.

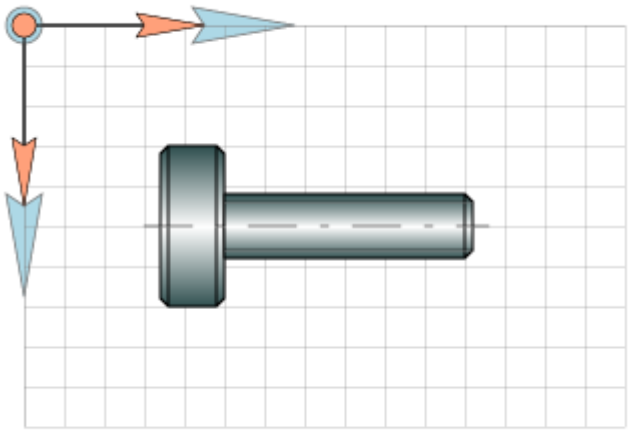

Figure 6-13. La vis et son système de coordonnées

Voici le code pour notre vis:

```
  <defs>
    <linearGradient              id="zylinderShade"       x2="0%"       y2="50%"
spreadMethod="reflect">
      <stop offset="0%" stop-color="darkslategray"/>
      <stop offset="100%" stop-color="white"/>
    </linearGradient>
    <g id="vis" stroke="black" stroke-linejoin="round" >
      <path fill="url(#zylinderShade)" d="M100,64 96,60 96,140 100,136 z
                                          M68,64 72,60 72,140 68,64 z
                                          M96,60 72,60 72,140 96,140 z" />
      <path fill="url(#zylinderShade)" d="M100,84 220,84 220,116 100,116 z
                                          M220,84 220,116 224,112 224,88 220,84
z" />
      <path stroke-width="0.4" fill="none" stroke-linecap="round"
            stroke-dasharray="24 12 4 12" d="M60,100 232,100" />
      <path stroke-width="0.4" fill="none" stroke-linecap="round"
            d="M100,88 224,88 M100,112 224,112" />
    </g>
  </defs>
```

Si nous référençons la vis avec `<use xlink:href="#vis" />`

le système local de coordonnées de l'instance de la vis (orange) coïncide avec le système de référence (bleu). Nous allons analyser les effets des valeurs de l'attribut 'transform'. Avec ce modèle, il devient clair que nous transformons en fait le système de coordonnées et non de simples objets graphiques.

Translation

La translation est un simple déplacement.

Syntaxe:

`translate(tx, ty)`

`tx` composante en x du vecteur de translation

`ty` composante en y du vecteur de translation

Nous appliquons une translation à l'instance du groupe 'vis', plus exactement à son système local de coordonnées.

```
<use xlink:href="#vis" transform="translate(100,130)" />
```

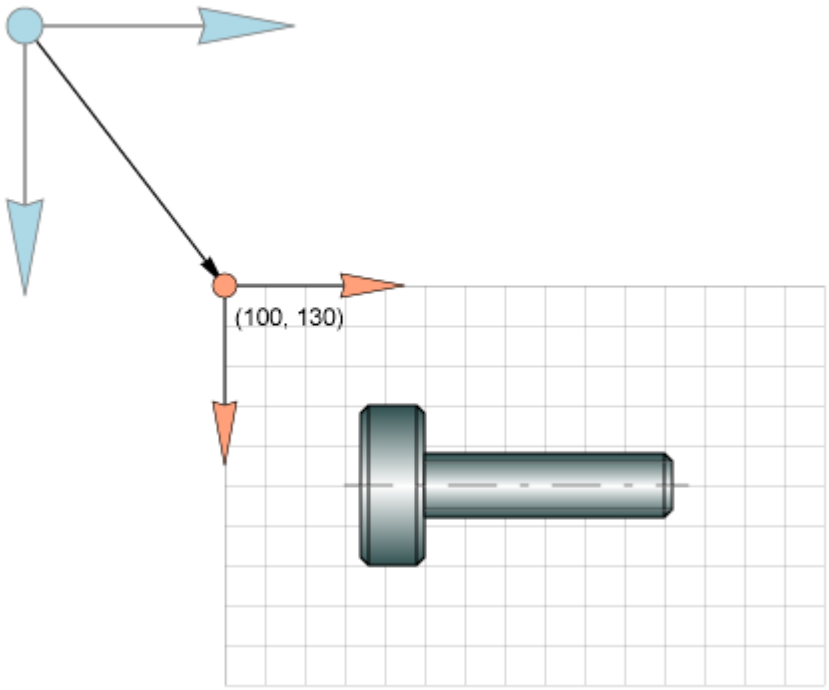

Figure 6-14.Translation de la vis

Facile! Vous pouvez vous demander quelle est la différence entre la translation et l'utilisation des attributs 'x' et 'y' pour placer un objet. Il semble bien que

```
<use xlink:href="#vis" transform="translate(100,130)" />
```

est identique à

```
<use xlink:href="#vis" x="100" y="130" />
```

Effectivement les deux donnent le même résultat. Alors pourquoi avons nous besoin des transformations? Un peu de patience, il y a d'autres transformations, nous pourrons les combiner.

Cependant, voyons les effets d'une transformation et de la donnée de valeurs à 'x' et 'y'

```
<use xlink:href="#vis" transform="translate(100,130)" x="130" y="-160" />
```

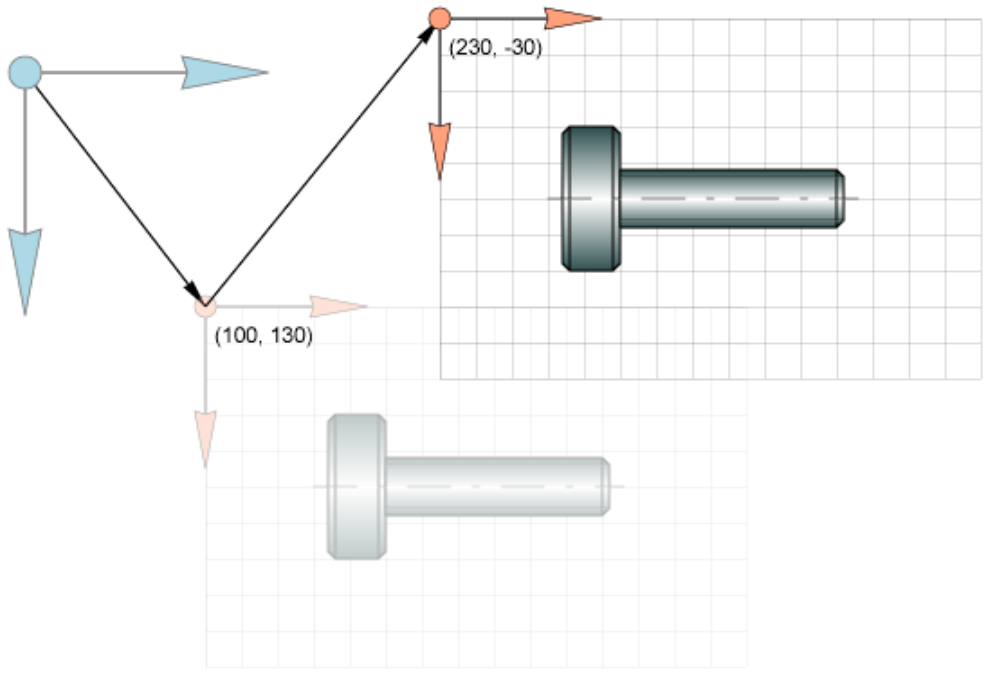

Figure 6-15. Translation et attributs x et y

L'effet que nous voyons est que la vis est d'abord déplacée par la translation, son origine vient en `(100,130)` et ensuite déplacée suivant les valeurs de 'x' et 'y' du vecteur `(130,-160)` pour avoir son origine en `(230,-30)`. Nous ajoutons simplement les valeurs de 'x' et 'y' au vecteur de la translation.

$$X_{local} = t_x + x$$
$$y_{local} = t_y + y$$

Dans ces formules nous pouvons changer l'ordre des termes.

$$X_{local} = x + t_x$$
$$y_{local} = y + t_y$$

donnera le même résultat. Nous pouvons également échanger les valeurs de 'x' et 'y' avec les composantes du vecteur de translation:

```
<use xlink:href="#vis" transform="translate(130,-160)" x="100" y="130" />
```

Nous pouvons trouver de multiples combinaisons pour amener notre instance de la vis au même endroit:

```
<use xlink:href="#vis" transform="translate(130,-160) translate(100,130)" />
<use xlink:href="#vis" transform="translate(100,130) translate(130,-160)" />
<use xlink:href="#vis" transform="translate(230,-30)" />
<use xlink:href="#vis" x="230" y="-30" />
```

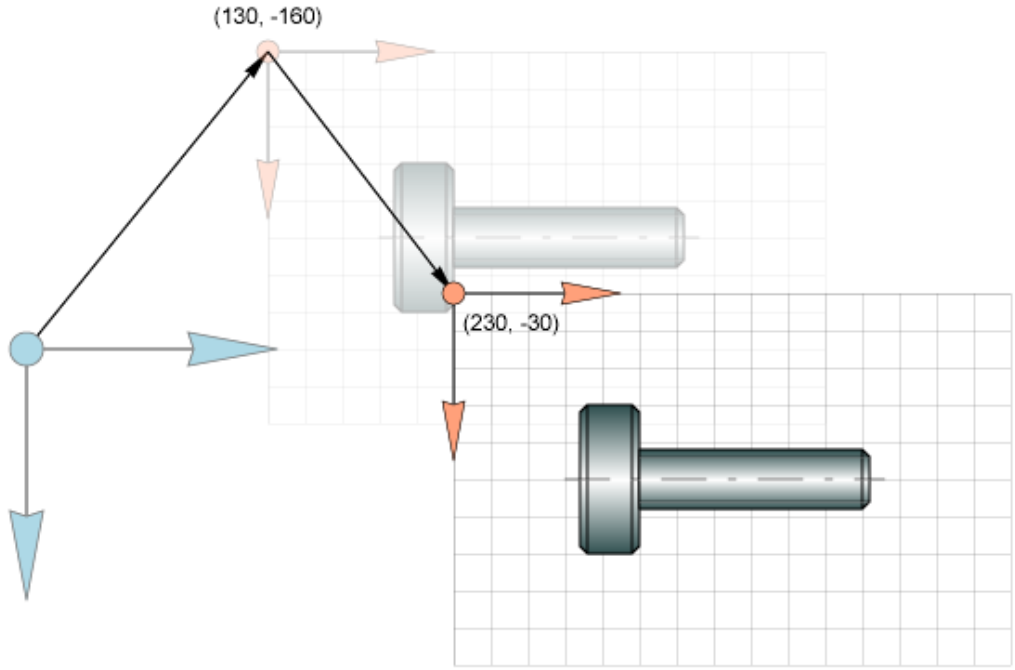

Figure 6-16. Pour arriver au même point (230,-30)

L'utilisation à la fois de la translation et des attributs 'x' et 'y' amène des confusions. Il est préférable de l'éviter. Ici l'ordre des translations ne changeait pas la position finale.
Avec d'autres transformations l'ordre changera la position finale, aussi que prendre en compte en premier, l'attribut 'transform' ou les attributs 'x' et 'y'?
Les spécifications SVG sont claires:

> *L'attribut 'transform' est appliqué à un élément avant de prendre en compte tout autre coordonnée ou longueur appliquée à cet élément.*

L'ordre à respecter est donc
Appliquer la transformation.
Appliquer les attributs 'x' et 'y'.

Rotation

La rotation fait tourner l'objet d'un certain angle autour du centre de rotation.

Syntaxe:

```
rotate(angle [, cx, cy])
angle angle de rotation en degrés
```

`cx` x du centre de rotation (optionnel – par défaut 0)
`cy` y du centre de rotation (optionnel – par défaut 0)

Nous appliquons la rotation à notre vis simplement avec

```
<use xlink:href="#vis" transform="rotate(25)" />
```

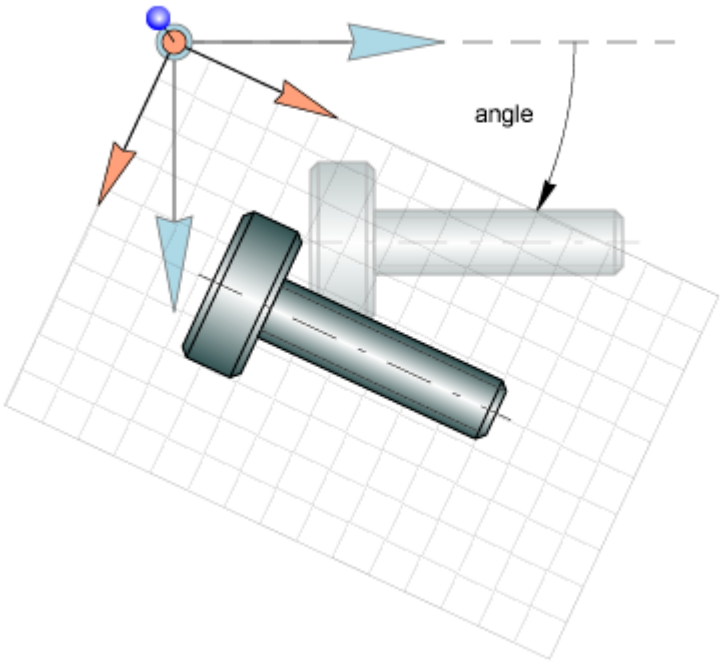

Figure 6-17. Rotation de la vis autour de l'origine

Nous avons une rotation de 25 degrés autour de l'origine du système de coordonnées de référence dans le sens des aiguilles d'une montre. Nous avons déjà noté que le sens positif en SVG n'est pas celui des mathématiciens en raison de l'orientation de l'axe des y vers le bas.

Utilisons les deux paramètres pour le centre de rotation, nous écrivons

```
<use xlink:href="#vis" transform="rotate(25, 100,100)" />
```

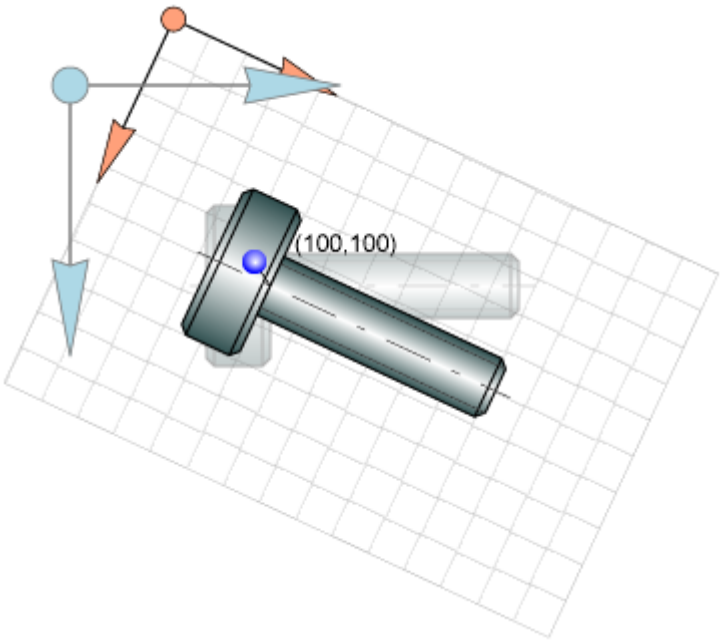

Figure 6-18. Rotation autour du point (100,100)

Nous imposons le point (`100,100`) marqué avec une épingle bleue comme centre de la rotation. Ce point est défini dans le repère de référence (repère bleu).

Résumons à propos de la rotation:

Le centre de la rotation est par défaut l'origine du repère de référence.
Si un point est défini avec les paramètres 'cx' et 'cy', il est alors le centre de la rotation.
L'angle de rotation est exprimé en degrés.
L'angle de rotation angle est positif ou négatif. Avec une valeur positive la rotation est dans le sens horaire.
Dans la rotation seul le centre est invariant (ne bouge pas).
Si l'angle est égal à zéro, nous avons la transformation 'identité', rien ne bouge.
La transformation inverse de `rotate(angle)` est `rotate(-angle)`. La transformation inverse est celle qui permet de revenir au point de départ, autrement dit d'annuler la transformation. En mathématiques nous dirions la transformation qui composée avec elle donne l'identité.

Homothétie, symétries et affinités

La transformation `scale` permet de changer la taille d'un objet. Si les facteurs sont les mêmes en x et en y, nous avons une homothétie. S'ils sont différents, d'un point de vue mathématique, nous avons la composée de deux affinités orthogonales qui modifie la taille en x et en y séparèment. Nous retrouverons également les symétries en jouant sur les valeurs 1 et –1.

Syntaxe:

`scale(sx [,sy])`

`sx` facteur pour x
`sy` facteur pour y, optionnel

Si le paramètre 'sy' n'est pas précisé, il est considéré comme égal à 'sx'.

```
<use xlink:href="#vis" transform="scale(2)" />
```

Le résultat est une vis dont la taille est le double de celle de l'original. Mais la position a également changé car le système de coordonnées local de la vis a subi la même transformation, l'unité est le double

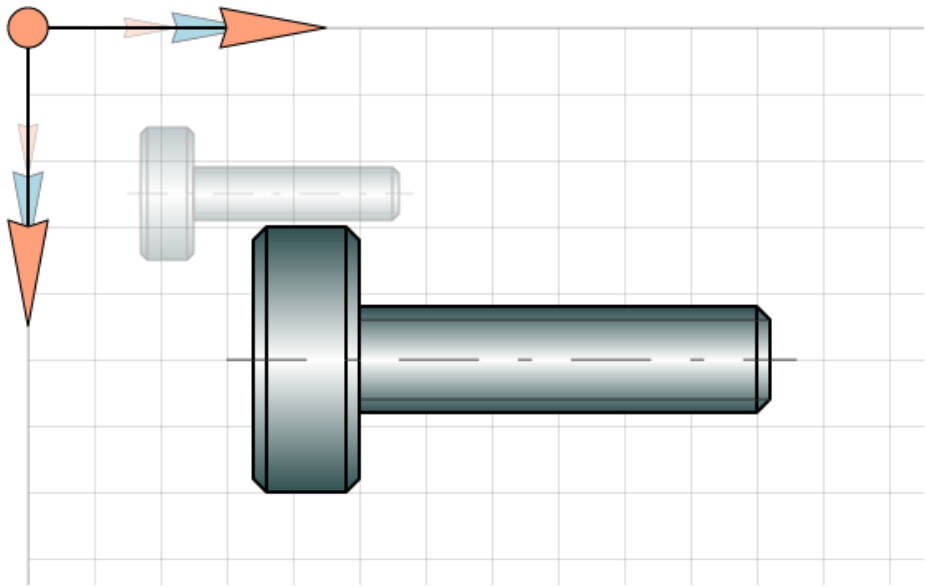

Figure 6-19. Multiplier la taille de la vis par deux

Nous pouvons évidemment réduire la taille de notre vis:

```
<use xlink:href="#vis" transform="scale(0.5)" />
```

Avec un facteur plus petit que un, nous diminuons la taille de la vis et comme l'unité est plus petite, la vis se rapproche de l'origine du repère. Un facteur nul réduirait toute la vis à un seul point.

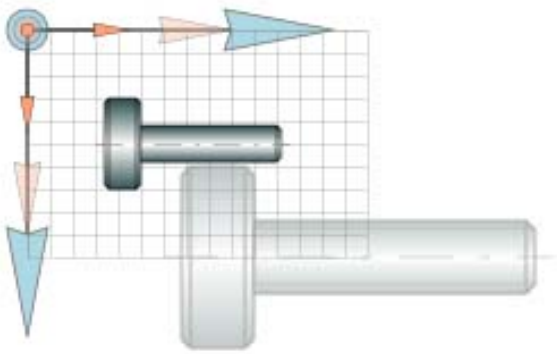

Figure 6-20. Diviser la taille de la vis par deux

Jusqu'à présent dans nos exemples les facteurs était égaux, nous avions donc des homothéties. Voyons le cas de facteurs différents en x et en y.

```
<use xlink:href="#vis" transform="scale(1,0.5)" />
```

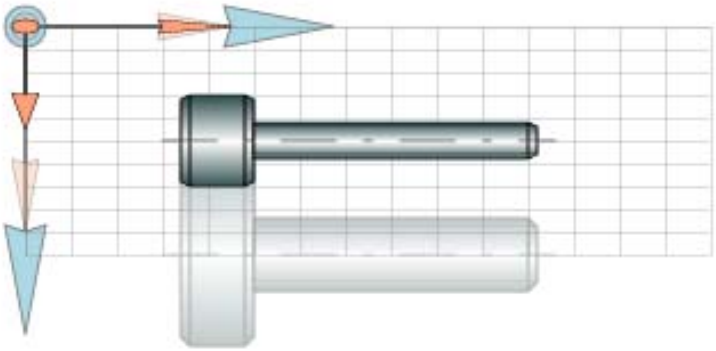

Figure 6-21. Scale(1,0.5) pour la vis

Ici, nous ne réduisons la taille de la vis qu'en y. Nous avons ici une affinité orthogonale axée sur l'axe des x et de rapport 0.5.

```
<use xlink:href="#vis" transform="scale(0.5,1)" />
```

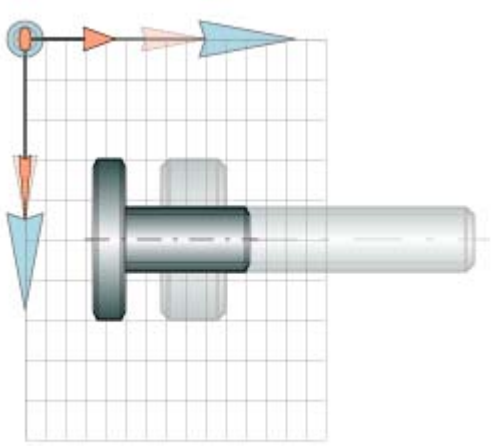

Figure 6-22. Scale(0.5,1) pour la vis

Ici, la taille de la vis n'est divisée par deux qu'en x. L'affinité est axée sur l'axe des y et a toujours un rapport de 0.5.

Les valeurs de 'sx' et 'sy' peuvent être négatives, nous pouvons retrouver des symétries:

```
<use xlink:href="#vis" transform="scale(-1,1)" />
```

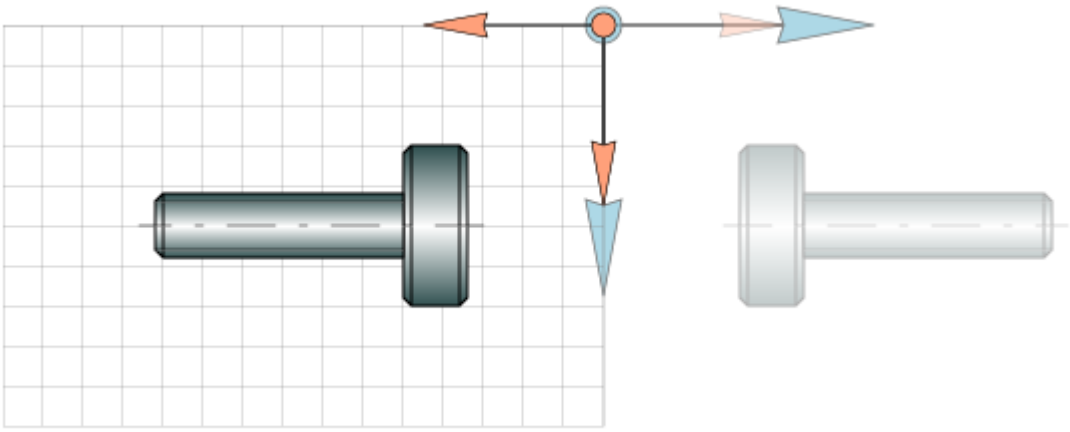

Figure 6-23. Symétrie par rapport à l'axe des y

Nous avons ici une symétrie par rapport à l'axe des y. Nous pouvons évidemment faire une symétrie par rapport à l'axe des x:

```
<use xlink:href="#vis" transform="scale(1,-1)" />
```

Et avec ces valeurs

```
<use xlink:href="#vis" transform="scale(-1,-1)" />
```

nous avons une symétrie par rapport à l'origine, ou encore une homothétie de rapport –1, ou encore la composée des deux symétries axiales précédentes, ou encore une rotation de 180 degrés.

```
<use xlink:href="#vis" transform="rotate(180)" />
```

Une remarque cependant, notre vis a un axe de symétrie et nous ne pouvons nous apercevoir que dans certaines de ces transformations, son orientation a changé. Quand vous vous

regardez dans un miroir, votre main droite est la main gauche de votre image, nous dirons que l'orientation a changé avec la symétrie par rapport au plan du miroir.

L'orientation change si et seulement si l'un des deux facteurs est négatif. Aussi les deux symétries axiales change l'orientation, alors que l'orientation est inchangée dans la symétrie centrale.

Résumons sur la transformation `scale`:

La taille de l'objet est uniformément modifiée si nous ne donnons que 'sx' ou deux valeurs égales (c'est une homothétie):

Un facteur supérieur à 1 ou inférieur à -1 agrandit l'objet.
Un facteur compris entre –1 et 1 diminue la taille de l'objet.
Avec un facteur nul, l'élément n'est pas rendu.
Les angles sont conservés, l'orientation également si le facteur est positif.
Un facteur de 1 nous donne l'identité – l'objet n'est pas modifié ni déplacé.
Un facteur de –1 nous donne la symétrie centrée à l'origine du repère.

La taille de l'objet est modifiée différemment suivant les x et les y si les deux valeurs sont différentes:

Les angles ne sont pas conservés

Nous retrouvons les symétries axées sur les axes des x et des y avec les facteurs –1 et 1

Les angles et longueurs sont conservés, l'orientation ne l'est pas.

Le centre de la transformation ets l'origine du repère de référence
La transformation inverse de `scale(`*`sx,sy`*`)` est la transformation `scale(`*`1/sx,1/sy`*`)`.

Glissements

Les transformations `skewX` et `skewY` n'ont pas d'équivalent mathématique simple.
Nous allons essayer de l'expliquer après avoir vu le résultat de cette transformation.

Syntaxe:

`skewX(angle])`
`angle` déplacement angulaire de l'axe des x en degrés

`skewY(angle])`
`angle` déplacement angulaire de l'axe des y en degrés

La valeur de l'angle doit être strictement comprise entre –90° et 90°

Voyons tout d'abord 'skewY':

```
<use xlink:href="#vis" transform="skewY(25)" />
```

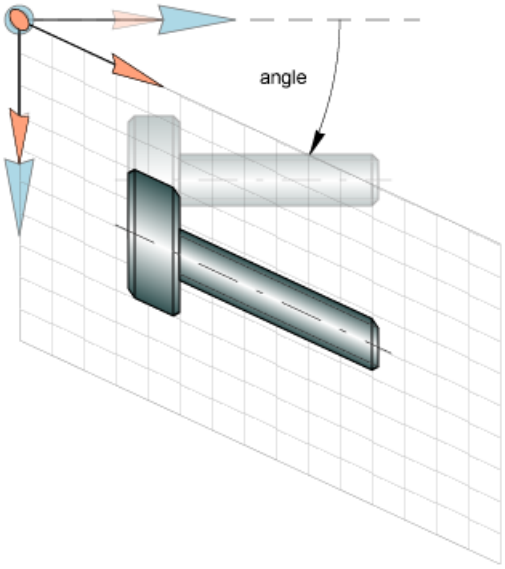

Figure 6-24. SkewY(25) pour la vis

Nous pouvons interpréter la transformation `skewY` comme une rotation de l'axe des x, alors que l'axe des y est inchangé. Une valeur de 90° détruirait le repère. Avec une valeur négative, l'axe tournerait dans l'autre sens.

La transformation `skewX` est similaire en échangeant le rôle des axes.

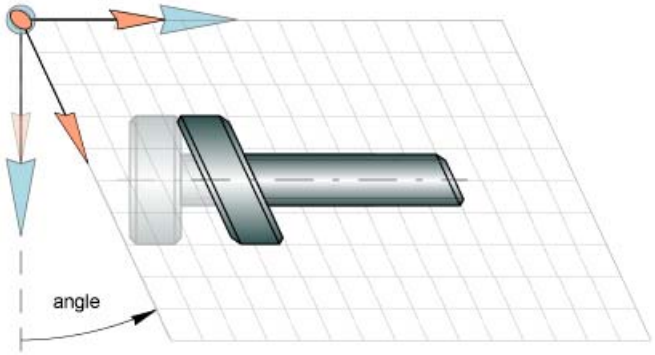

Figure 6-25. SkewX(25) pour la vis

La transformation `skewX` consiste en une rotation de l'axe des y, tandis que l'axe des x est inchangé.

Nous ne pouvons assimiler ces transformations à des rotations.
Nous pouvons comprendre la transformation `skewX` comme un déplacement des points dans la direction des x, mais le déplacement n'est pas le même pour tous les points, il dépend de leur y, plus y est grand, plus le déplacement est important.

Pour définir mathématiquement cette transformation, nous définissons les rapports

$$k_x = \frac{\Delta x}{y}; k_y = \frac{\Delta y}{x}$$

Il y a une relation entre ces rapports et l'angle de la transformation

$$\tan\alpha_x = k_x ; \tan\alpha_y = k_y$$

Nous retrouverons ces tangentes quand nous exprimerons ces transformations par une matrice.

Résumons pour cette transformation:

Un objet peut être déplacé indépendamment dans les directions des x ou des y avec les transformations `skewX` et `skewY` .
L'angle est donné en degrés, il peut être positif ou négatif.
L'angle est inférieur à 90° et supérieur à –90°
La transformation `skewX` ne préserve que les longeurs parallèles à l'axe des x.
La transformation `skewY` ne préserve que les longeurs parallèles à l'axe des y.
Les angles ne sont pas préservés.
Dans `skewX` l'axe des y est invariant.
Dans `skewY` l'axe des x est invariant.
Pour `angle = 0` nous avons l'identité pour `skewX` et `skewY`.
La transformation inverse de `skewX(angle)` est `skewX(-angle)` (de même pour `skewY`).

Composition de transformations

Nous pouvons composer ces transformations élémentaires dans l'attribut 'transform'.

Syntaxe:

```
transform="trfN ... trf2 trf1"
```

`trf1` première transformation élémentaire
`trf2` seconde transformation élémentaire
`trfN` Nième transformation élémentaire

Mais attention l'ordre des transformations est très important, il modifie le résultat.

Les transformations de l'attribut `transform` sont prises en compte de droite à gauche.

Complétons notre collection d'objets.

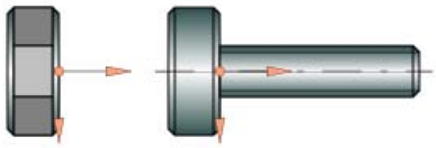

Figure 6-26. Nos objets

A notre boulon nous ajoutons un écrou. Les deux objets sont décrits géométriquement dans leur repère local dessiné en orange. Nous voulons utiliser ces deux objets pour assembler deux plateaux.

Attention, cette fois le boulon est décrit dans un autre repère que précédemment.

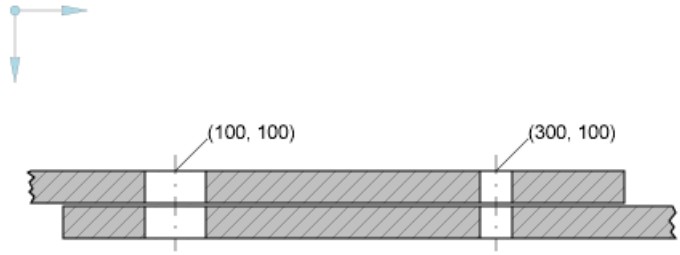

Figure 6-27. Deux plateaux à assembler

Voici le code de tous ces objets

```
  <defs>
    <linearGradient              id="zylinderShade"       x2="0%"       y2="50%"
spreadMethod="reflect">
      <stop offset="0%" stop-color="darkslategray"/>
      <stop offset="100%" stop-color="white"/>
    </linearGradient>
    <pattern id="hatch" width="10" height="10" patternUnits="userSpaceOnUse">
      <rect width="10" height="10" stroke="none" fill="silver" />
      <polyline points="0,10 10,0" stroke="gray" stroke-width="0.25"/>
    </pattern>
    <g id="boulon" stroke="black" stroke-linejoin="round" >
      <path fill="url(#zylinderShade)" d="M0,-36 -4,-40 -4,40 0,36 z
                                          M-32,-36 -28,-40 -28,40 -32,36 z
                                          M-4,-40 -28,-40 -28,40 -4,40 z" />
      <path fill="url(#zylinderShade)" d="M0,-16 120,-16 120,16 0,16 z
                                          M120,-16 120,16 124,12 124,-12 120,-
16 z" />
      <path stroke-width="0.4" fill="none" stroke-linecap="round"
            stroke-dasharray="24 12 4 12" d="M-40,0 132,0" />
      <path stroke-width="0.4" fill="none" stroke-linecap="round"
            d="M0,-12 124,-12 M0,12 124,12" />
    </g>
     <g id="ecrou" stroke="black" stroke-linejoin="round">
        <path fill="url(#zylinderShade)" d="M0,-36 -4,-40 -4,40 0,36 z
                                            M-32,-36 -28,-40 -28,40 -32,36 z
                                            M-4,-40 -28,-40 -28,40 -4,40 z" />
        <path fill="gray" d="M-4,-40 -28,-40 -28,-16 -4,-16 z" />
        <path fill="silver" d="M-4,-16 -28,-16 -28,16 -4,16 z" />
        <path fill="gray" d="M-4,16 -28,16 -28,40 -4,40 z" />
     </g>
    <g id="plateaux">
      <path id="dessus" fill="url(#hatch)" stroke="black"
            d="M10,100  81,100  81,120  10,120  12,116  9,113  11,111  9,107  12,104
8,101 z
               M81,100 119,100
               M81,120 119,120
               M119,100 290,100 290,120 119,120 z
               M290,100 310,100
               M290,120 310,120
               M310,100 380,100 380,120 310,120 z"/>
      <path id="dessous" fill="url(#hatch)" stroke="black"
            d="M30,122 81,122 81,142 30,142 z
               M81,122 119,122
               M81,142 119,142
               M119,122 290,122 290,142 119,142 z
               M290,122 310,122
               M290,142 310,142
               M310,122 410,122 412,126 408,131 411,135 409,138 412,142 310,142
z"/>
      <path stroke="black" stroke-width="0.4" fill="none" stroke-dasharray="12
6 2 6"
            d="M100,90 100,150 M300,90 300,150" />
      <path stroke="black" stroke-width="0.5" fill="none"
            d="M100,100 120,80 M300,100 320,80" />
      <text x="120" y="80" text-anchor="start">(100, 100)</text>
```

```
      <text x="320" y="80" text-anchor="start">(300, 100)</text>
    </g>
  </defs>
```

La géométrie des plateaux est décrite dans le repère de référence (bleu). Nous plaçons la première instance du boulon avec:

```
<use xlink:href="#boulon" />
```

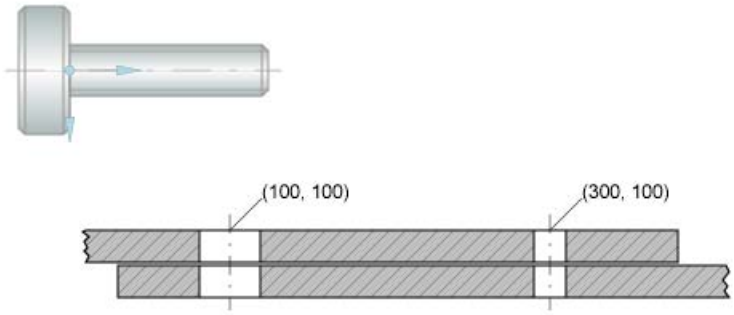

Figure 6-28. Les plateaux et le boulon

Le repère local du boulon coïncide avec le repère de référence. Nous devons lui appliquer une série de transformations pour le mettre en place. Nous devons

- faire tourner le boulon de 90 degrés.
- le déplacer par une translation au point (100,100).

Nous écrirons donc, sachant que les transformations agiront de droite à gauche:

```
<use xlink:href="#boulon" transform="translate(100,100) rotate(90)" />
```

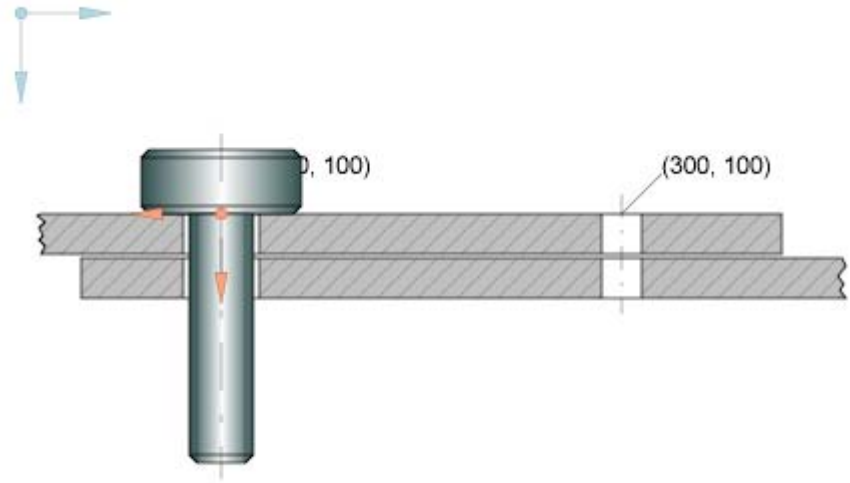

Figure 6-29. Le boulon est en place

Ceci fonctionne bien. Cela semble facile, mais nous allons voir de plus près en dessinant quelques étapes de la transformation.

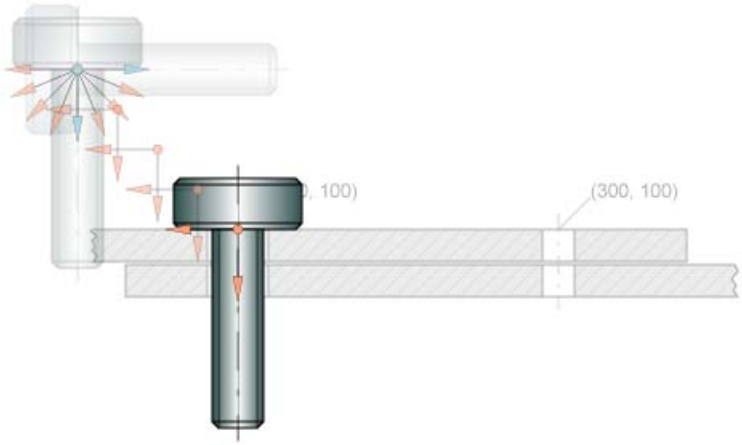

Figure 6-30. Etapes de la mise en place du boulon

L'ordre des transfromations est essentiel. Voyons ce qui se passe en changeant leur ordre.

```
<use xlink:href="#boulon" transform="rotate(90) translate(100,100)" />
```

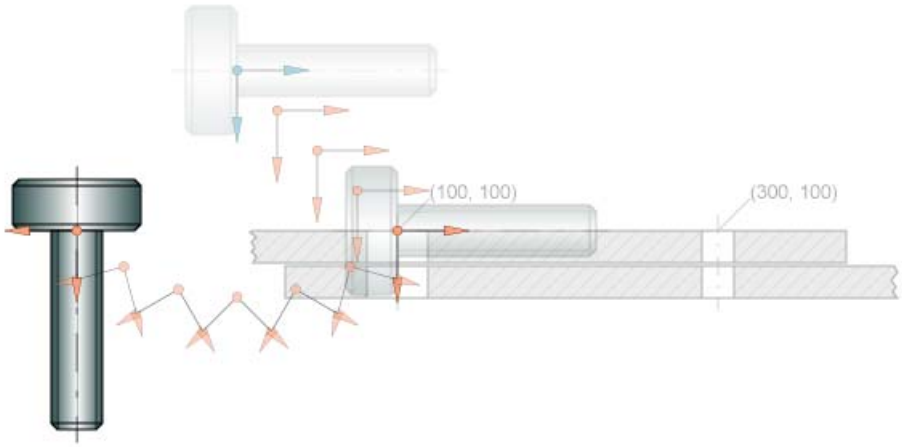

Figure 6-31. Les transformations dans l'ordre contraire

Ici, le boulon n'est pas en place, car la rotation s'applique quand le boulon est déjà déplacé et étant centrée à l'origine du repère de référence, elle déplace le boulon en le faisant tourner.

> *Le centre par défaut de la rotation est l'origine du repère de l'élément parent.*

Nous pouvons corriger en donnant explicitement le centre de la rotation à l'origine du repère local du boulon après sa translation.

```
   <use xlink:href="#boulon" transform="rotate(90, 100,100) translate(100,100)"
/>
```

Nous pouvons maintenant mettre en place notre boulon.
Une rotation de –90°
Une translation au point `(100,142)`.

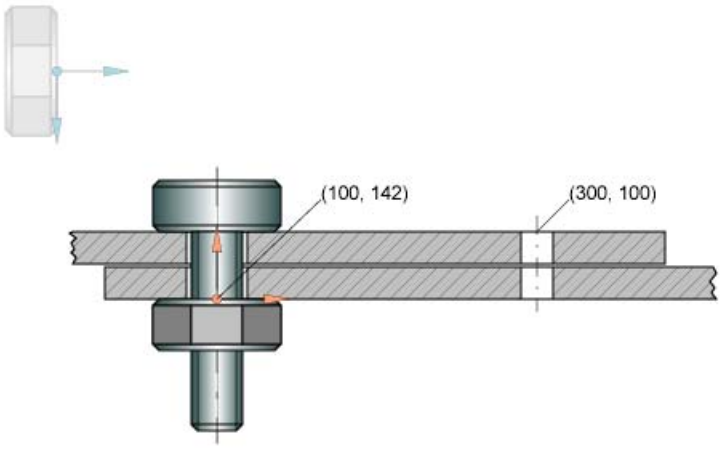

Figure 6-32. Les plateaux avec un boulon en place

```
 <use xlink:href="#ecrou" transform="translate(100,142) rotate(-90)"
/>
```

ou encore

```
   <use xlink:href="#ecrou" transform="rotate(-90, 100,142) translate(100,142)"
/>
```

Ceci fonctionne très bien, quoique le boulon paraisse un peu disproportionné.

Passons à l'autre boulon, mais nous réalisons que l'autre trou a un diamètre moitié de celui du premier. Nous devons donc prendre des boulons plus petits.

Nous pouvons changer la taille de notre boulon avec les transformations. Nous avons en fait une infinité de boulons avec des tailles différentes.

Par où commencer? Pour diviser la taille de notre boulon par deux, nous utilisons `scale(0.5)`. Mais nous devons également faire tourner et déplacer notre boulon. Dans quel ordre appliquez ces trois transformations?

Pour 'scale', le centre de la transformation est l'origine du repère de référence. Comme le repère du boulon coïncide avec ce repère, nous pouvons commencer par cette transformation.

```
<use xlink:href="#boulon" transform="translate(300,100) rotate(90) scale(0.5)"
/>
```

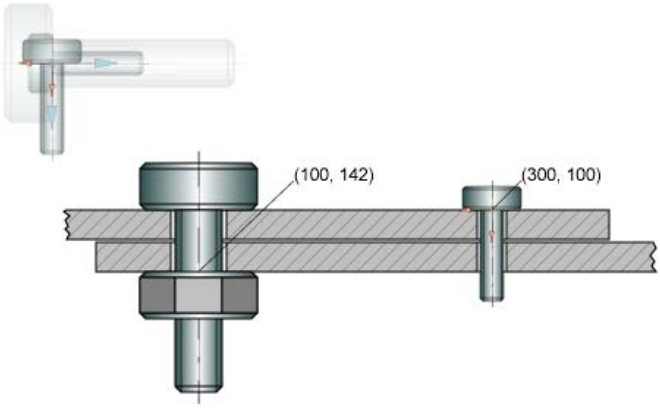

Figure 6-33. Un autre boulon en place

Nous avons le résultat escompté. Pour le second écrou nous permutons 'scale' et 'rotate' pour voir …

```
<use xlink:href="#nut" transform="translate(300,142) scale(0.5) rotate(-90)" />
```

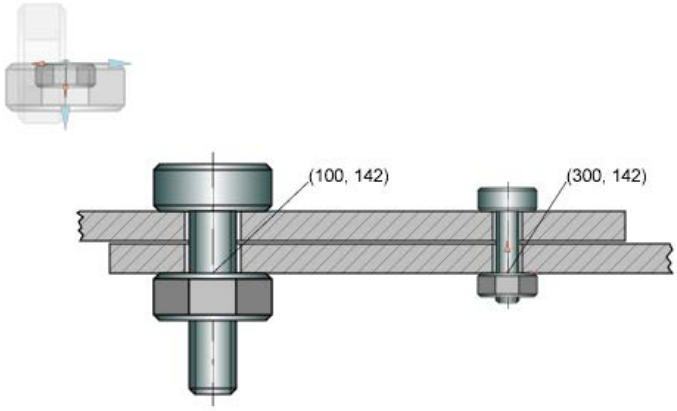

Figure 6-34. Les deux boulons en place

Très bien, les deux plateaux sont solidement fixés. Mais soyons prudent(e)s avec l'ordre des transformations, surtout si les rapports de 'scale' ne sont pas égaux.

Illustrons ceci par un exemple simple:

```
<use xlink:href="#vis" transform="scale(0.5,1) rotate(90)" />
<use xlink:href="#vis" transform="rotate(90) scale(0.5,1)" />
```

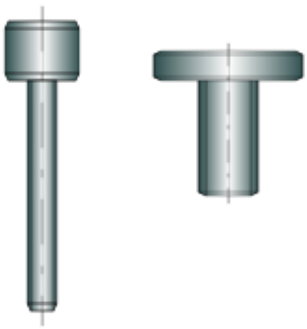

Figure 6-35. Ordre des transformations

Que de boulons de tailles différentes! Serait-il possible d'avoir deux boulons de même diamètre, de même tête mais avec des longeurs différentes? Avec SVG 1.0 et un seul groupe, cela ne semble pas possible.

Terminons avec le code de l'assemblage final:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
"http://www.w3.org/TR/SVG/DTD/svg10.dtd">
<svg width="100%" height="100%" xmlns="http://www.w3.org/2000/svg">
  <defs>
    <!-- boulon, écrou et plateaux -->
  </defs>
```

```
  <g transform="translate(60,60)">
    <use xlink:href="#plateaux" />
    <use xlink:href="#boulon" transform="translate(100,100) rotate(90)" />
    <use xlink:href="#ecrou" transform="translate(100,142) rotate(-90)" />
    <use    xlink:href="#boulon"    transform="translate(300,100)    rotate(90)
scale(0.5)" />
    <use xlink:href="#ecrou" transform="translate(300,142) scale(0.5) rotate(-
90) " />
  </g>
</svg>
```

Transformations emboîtées

Nous venons de voir l'application de transformations multiples à un même élément. Nous devons maintenant analyser comment les transformations appliquées au container agissent avec les transformations appliquées aux éléments du contenu. Nous prenons un exemple où des transformations sont appliquées à l'original et aux instances de l'objet.

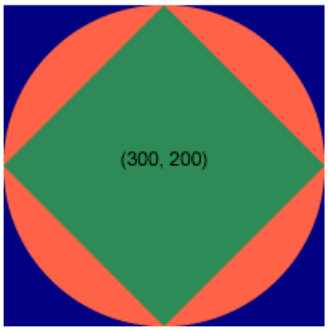

Figure 6-36. Deux carrés et un cercle

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
          "http://www.w3.org/TR/SVG/DTD/svg10.dtd">
<svg width="100%" height="100%" xmlns="http://www.w3.org/2000/svg">
  <defs>
    <rect id="inner" width="200" height="200" fill="seagreen"
          transform="rotate(45) translate(-100,-100)" />
  </defs>
  <g transform="translate(300,200)">
    <rect width="100" height="100" fill="navy" stroke-width="2"
          transform="scale(2) translate(-50,-50)" />
    <circle r="100" stroke="none" fill="tomato" />
    <use xlink:href="#inner" transform="scale(0.707)" />
  </g>
</svg>
```

Nous avons trois éléments dans ce document SVG document (le texte des coordonnées mis à part) – deux carrés et un cercle.
Le carré bleu marine (`navy`) est défini avec son angle supérieur gauche en (0,0), nous ne donnons pas de valeur à 'x' et 'y'. Il est ensuite translaté pour que son centre coïncide avec l'origine du repère de référence. Enfin sa taille est multipliée par deux.
Le cercle orange foncé (`tomato`) est centré en `(0,0)` origine du repère de référence.
Le carré vert foncé (`seagreen`) est défini dans la section <defs>. Il a une taille double de celle du carré bleu marine. Il est ensuite translaté pour que son centre coïncide avec l'origine. Il est ensuite tourné de 45° et enfin sa taille est multipliée par 0.707 pour qu'il soit inscrit dans le cercle.

Ces trois éléments sont placés dans un groupe qui est lui-même translaté au centre du document en `(300,200)`.

Nous allons créer la même image avec une structure élémentaire sans utiliser <defs>, <g> ou <use>.

Nous commençons avec le cercle. Il suffit d'appliquer la transformation de groupe à l'élément lui-même.

<circle r="100" stroke="none" fill="tomato" **transform="translate(300,200)"** />

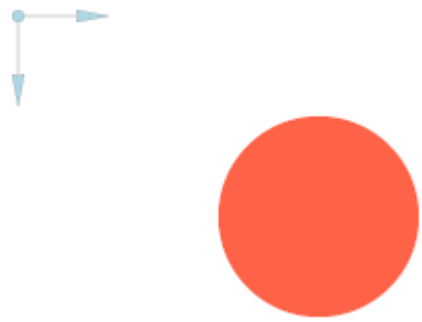

Figure 6-37. Le cercle est en place

Passons au carré bleu marine. Cet élément a ses propres transformations, nous y ajoutons **au début** la transformation du groupe.

```
<rect width="100" height="100" fill="navy" stroke-width="2"
      transform="translate(300,200) scale(2) translate(-50,-50)" />
```

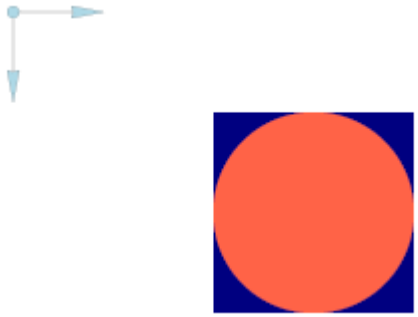

Figure 6-38. Cercle et carré bleu marine sont en place

Nous pouvons généraliser ceci. Si nous avons la structure suivante:

```
<g transform="t3">
  <g transform="t2">
    <g transform="t1">
      <element transform="t0" />
    </g>
  </g>
</g>
```

Le résultat sera le même qu'avec:

```
<element transform="t3 t2 t1 t0" />
```

Nous devons composer les transformations en partant de l'élément et en remontant les groupes parents. Nous pouvoir aussi voir ceci en terme d'héritage:

> *Un élément hérite des transformations de son parent qui lui sont appliquées après ses propres transformations.*

Enfin le carré vert foncé. Nous devons d'abord éliminer l'élément <use> en appliquant à l'élément la transformation utilisée par l'élément 'use':

<rect id="inner" width="200" height="200" fill="seagreen"
transform=" **scale(0.707)** rotate(45) translate(-100,-100)" />

Notez que là aussi, la transformation est placée **au début**, donc elle sera appliquée **après** les transformations de l'élément original. Enfin nous extrayons l'élément de son groupe en ajoutant la translation de groupe au début comme pour le carré bleu marine.

<rect id="inner" width="200" height="200" fill="seagreen"
transform=" **translate(300,200)** scale(0.707) rotate(45) translate(-100,-100)" />

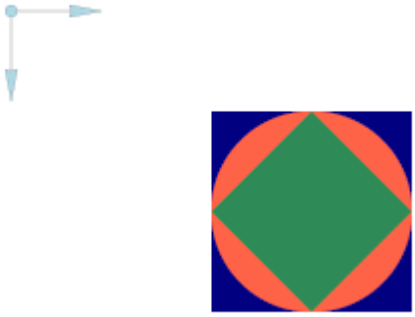

Figure 6-39. Les trois objets en place

Nous pouvons généraliser pour des instances successives (ou des groupes).

<*element* id="e0" transform="*t0*" />
<use id="e1" xlink:href="#e0" transform="*t1*" />
<use id="e2" xlink:href="#e1" transform="*t2*" />
<use id="e3" xlink:href="#e2" transform="*t3*" />

est identique à

<element id="e3" transform="t3 t2 t1 t0" />

En termes d'héritage nous pouvons dire:

> *Une instance particulière d'un élément hérite des transformations de l'élément 'use' qui seront appliquées après les transformations de l'élément.*

Voici la version de notre document sans structure.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
"http://www.w3.org/TR/SVG/DTD/svg10.dtd">
<svg width="100%" height="100%" xmlns="http://www.w3.org/2000/svg">
  <rect width="100" height="100" fill="navy" stroke-width="2"
        transform="translate(300,200) scale(2) translate(-50,-50)"/>
  <circle r="100" stroke="none" fill="tomato" transform="translate(300,200)" />
  <rect id="inner" width="200" height="200" fill="seagreen"
        transform="translate(300,200) scale(0.707) rotate(45) translate(-100,-
100)" />
</svg>
```

Nous allons appliquer ces règles pour créer une fractale, le triangle de Sierpinski. Nous partons d'un simple triangle.

```
<?xml version="1.0" ?>
<svg width="600" height="400">
   <defs>
      <polygon id="e0" fill="navy" points="0,0 200,0 100,-200" />
   </defs>
   <use xlink:href="#e0" transform="translate(200,300)" />
</svg>
```

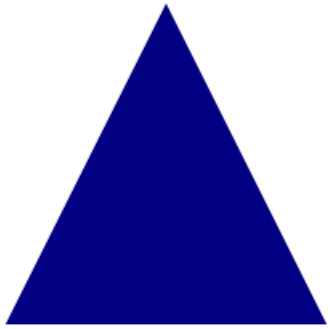

Figure 6-40. Le triangle de départ

Nous réutilisons trois fois ce triangle en divisant sa taille par 2 et en positionnant les triangles au sommet du triangle de départ.

```
<?xml version="1.0" ?>
<svg width="600" height="400">
   <defs>
      <polygon id="e0" fill="navy" points="0,0 200,0 100,-200" />
      <g id="e1">
         <use xlink:href="#e0" transform="translate(0,0) scale(0.5)" />
         <use xlink:href="#e0" transform="translate(100,0) scale(0.5)"
/>
         <use xlink:href="#e0" transform="translate(50,-100) scale(0.5)"
/>
      </g>
   </defs>
   <use xlink:href="#e1" transform="translate(200,300)" />
</svg>
```

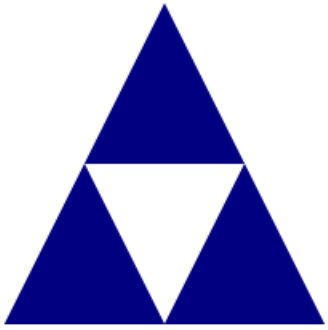

Figure 6-41. Trois triangles

Nous répétons cette opération simplement en collant le groupe e1 et en renommant e1, e2, e3 ….

```
   <?xml version="1.0" ?>
   <svg width="600" height="400">
      <defs>
         <polygon id="e0" fill="navy" points="0,0 200,0 100,-200" />
         <g id="e1">
            <use xlink:href="#e0" transform="translate(0,0)     scale(0.5)"
/>
            <use xlink:href="#e0" transform="translate(100,0)   scale(0.5)"
/>
            <use xlink:href="#e0" transform="translate(50,-100) scale(0.5)"
/>
         </g>
         <g id="e2">
           <use xlink:href="#e1" transform="translate(0,0)      scale(0.5)"
/>
           <use xlink:href="#e1" transform="translate(100,0)    scale(0.5)"
/>
           <use xlink:href="#e1" transform="translate(50,-100)  scale(0.5)"
/>
         </g>
         <g id="e3">
           <use xlink:href="#e2" transform="translate(0,0)      scale(0.5)"
/>
           <use xlink:href="#e2" transform="translate(100,0)    scale(0.5)"
/>
           <use xlink:href="#e2" transform="translate(50,-100)  scale(0.5)"
/>
         </g>
         <g id="e4">
           <use xlink:href="#e3" transform="translate(0,0)      scale(0.5)"
/>
           <use xlink:href="#e3" transform="translate(100,0)    scale(0.5)"
/>
           <use xlink:href="#e3" transform="translate(50,-100)  scale(0.5)"
/>
         </g>
         <g id="e5">
           <use xlink:href="#e4" transform="translate(0,0)     scale(0.5)"
/>
           <use xlink:href="#e4" transform="translate(100,0)   scale(0.5)"
/>
           <use xlink:href="#e4" transform="translate(50,-100) scale(0.5)"
/>
         </g>
      </defs>
      <use xlink:href="#e5" transform="translate(200,300)" />
   </svg>
```

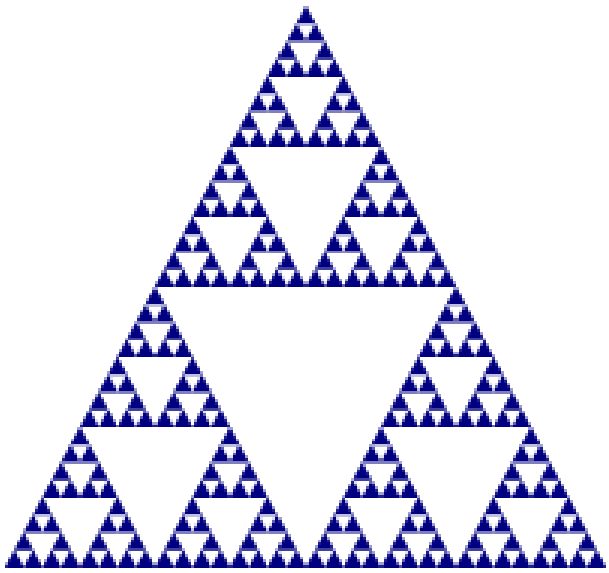

Figure 6-42. Quelques étapes supplémentaires

Nous pouvons appliquer cette méthode à d'autres objets pour obtenir des fractales de type "*iterated function systems*" ("*IFS*"). Nous en verrons d'autres au chapitre 12, en particulier la courbe de Von Koch.

Matrices et transformations

L'attribut 'transform' supporte un dernier type de transformation, le plus général, une transformation représentée par une matrice.

Syntaxe:

```
matrix(a, b, c, d, e, f)
```

`a, b, c, d, e, f` nombres formant la matrice

Reprenons notre exemple de plateaux à assembler.

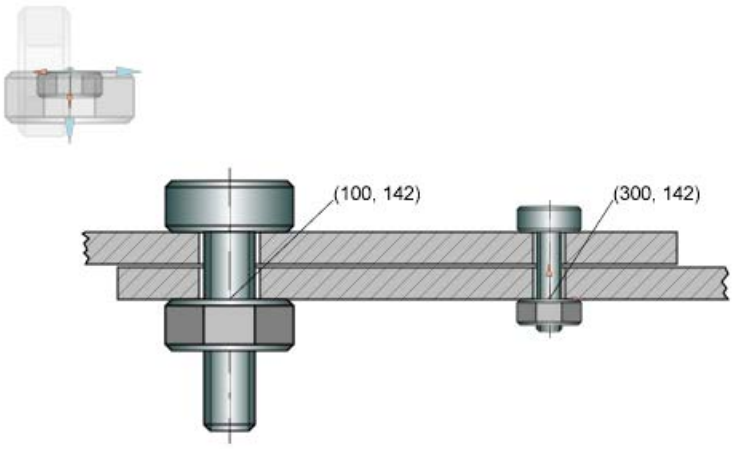

Figure 6-43. Les plateaux assemblés

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
"http://www.w3.org/TR/SVG/DTD/svg10.dtd">
<svg width="100%" height="100%" xmlns="http://www.w3.org/2000/svg">
  <defs>
    <!-- boulon, écrou et plateaux sont définis ici -->
  </defs>
  <g transform="translate(60,60)">
    <use xlink:href="#plateaux" />
    <use xlink:href="#boulon" transform="matrix(0,1,-1,0,100,100)" />
    <use xlink:href="#ecrou" transform="matrix(0,-1,1,0,100,142)" />
    <use xlink:href="#boulon" transform="matrix(0,0.5,-0.5,0,300,100)" />
    <use xlink:href="#ecrou" transform="matrix(0,-0.5,0.5,0,300,142)" />
  </g>
</svg>
```

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Nous devons faire un peu de calcul matriciel. Une matrice est un tableau de nombres. En SVG les matrices auront trois lignes et trois colonnes (3x3) pour pouvoir représenter les transformations. Les coordonnées de dimension 2 sont complétées par un 1 pour former un vecteur de dimension 3, afin de pouvoir gérer les translations.
Nous avons ici ce que l'on nomme des coordonnées homogènes. Les nombres des deux premières lignes sont les composants de la matrice, la dernière ligne est toujours `(0 0 1)`, nous pouvons l'oublier.

Comment calculer avec cette matrice les coordonnées de l'image du point (x,y), soit x' et y' ?

Nous multiplions les lignes de la matrice par la colonne (x y 1) terme à terme et en faisant la somme des produits obtenus.

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a \cdot x + c \cdot y + e \\ b \cdot x + d \cdot y + f \\ 1 \end{pmatrix}$$

Les coordonnées de l'image `(x',y')` du point `(x,y)` seront `(ax+cy+e, bx+dy+f)`.
x' et y' sont des fonctions linéaires de x et y, ces transformations seront dites *affines.*

Matrice de translation

la matrice de la translation se présente ainsi

$$T_{trans} = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix}$$

`tx` et `ty`.étant les valeurs données au vecteur de translation avec `translate(tx,ty)`.

Matrice de rotation

La matrice de rotation utilise quelques fonctions trigonométriques

$$T_{rotate} = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Alpha est l'angle de la rotation en **radians**. Avec `rotate(angle)` l'angle est exprimé en degrés, aussi nous devons faire une conversion avec $\alpha = angle \cdot \frac{\pi}{180}$

Dans cet exemple, le centre de la rotation est en (0,0).

Matrice pour 'scale'

$$T_{scale} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Les facteurs `sx` *et* `sy` sont ceux définis dans `scale(sx,sy)`.

Matrice pour skewX et skewY

$$T_{skewX} = \begin{pmatrix} 1 & \tan\alpha & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}; T_{skewY} = \begin{pmatrix} 1 & 0 & 0 \\ \tan\alpha & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Le facteur `tan(alpha)` doit être calculé avec l'angle en radians. Nous devons faire la même conversion que pour la rotation à partir de l'angle en degrés donné par `skewX(angle)` ou `skewY(angle)`.

Composer les matrices de transformations

Les transformations sont composées en multipliant leurs matrices, cette multiplication n'est pas commutative, l'ordre des matrices est très important. Pour faire le produit des matrices, nous multiplions lignes de l'une par colonnes de l'autre:

$$T = \begin{pmatrix} a_1 & c_1 & e_1 \\ b_1 & d_1 & f_1 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} a_2 & c_2 & e_2 \\ b_2 & d_2 & f_2 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} a_1 \cdot a_2 + c_1 \cdot b_2 & a_1 \cdot c_2 + c_1 \cdot d_2 & a_1 \cdot e_2 + c_1 \cdot f_2 + e_1 \\ b_1 \cdot a_2 + d_1 \cdot b_2 & b_1 \cdot c_2 + d_1 \cdot d_2 & b_1 \cdot e_2 + d_1 \cdot f_2 + f_1 \\ 0 & 0 & 1 \end{pmatrix}$$

Si nous appliquons au calcul des coordonnées de l'image du point (x,y), nous pouvons remplacer les deux matrices par leur produit. Attention, la matrice de droite représente la première transformation à appliquer au point.

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a_1 & c_1 & e_1 \\ b_1 & d_1 & f_1 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} a_2 & c_2 & e_2 \\ b_2 & d_2 & f_2 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Les matrices sont ainsi dans le même ordre que les transformations élémentaires dans l'attribut 'transform'.

Exemple de calcul matriciel

Mettons en pratique pour notre exemple de plateaux à assembler.

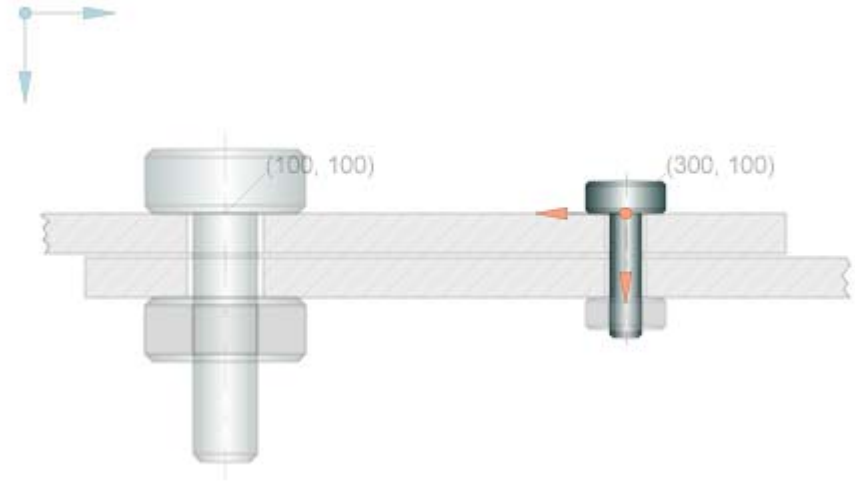

Figure 6-44. Mise en place avec une matrice

Nous positionnons le boulon avec

```
<use xlink:href="#boulon" transform="translate(300,100) rotate(90) scale(0.5)"
/>
```

Nous multiplions les matrices dans le même ordre

$$T = \begin{pmatrix} 1 & 0 & 300 \\ 0 & 1 & 100 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \cos 90° & -\sin 90° & 0 \\ \sin 90° & \cos 90° & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0.5 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Comme sin 90° = 1 et cos 90° = 0, nous pouvons simplifier:

$$T = \begin{pmatrix} 1 & 0 & 300 \\ 0 & 1 & 100 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0.5 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Commençons à multiplier les matrices. Ne vous étonnez pas si le calcul commence avec les deux matrices de gauche, l'ordre des matrices est important, mais l'ordre des opérations ne change pas le résultat (associativité du produit de matrices)
Le produit des deux matrices de gauche est fait:

$$T = \begin{pmatrix} 0 & -1 & 300 \\ 1 & 0 & 100 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0.5 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Nous terminons le produit des matrices

$$T = \begin{pmatrix} 0 & -0.5 & 300 \\ 0.5 & 0 & 100 \\ 0 & 0 & 1 \end{pmatrix}$$

et voici la matrice qui représente la transformation composée.
Seuls les nombres des deux premières lignes sont à considérer:

$T = \begin{pmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{pmatrix}$ se traduit par `matrix(a, b, c, d, e, f)`

Nous pouvons donc remplacer les trois transformations par une seule transformation définie par la matrice produit

```
<use xlink:href="#boulon" transform="matrix(0,0.5,-0.5,0,300,100)" />
```

Facile! Non? Pourquoi tant de calculs:

> *En SVG, les attributs des transformations sont convertis en matrices pour les calculs de rendu. Aussi l'utilisation de matrices donne de meilleures performances.*

Je vous recommande l'utilisation des matrices pour la génération de code SVG (côté serveur ou côté client) et pour l'animation SMIL.

Note du traducteur: La transformation définie par une matrice ne peut être utilisée avec animateTransform ce qui ôte beaucoup d'intérêt aux matrices pour l'animation si nous n’utilisons pas de script.