# Tutoriel sur Cairo pour les programmeurs Python

Cairo est une puissante bibliothèque graphique 2D.

Ce document vous présente la façon dont fonctionne Cairo et la plupart des fonctions que vous utiliserez pour créer le graphisme que vous désirez.

Afin de suivre ce tutoriel sur votre ordinateur, vous avez besoin des éléments suivants:

1. Cairo lui-même,
2. Python pour exécuter les morceaux de code, et
3. Pycairo pour joindre les deux précédents.

> Ce tutoriel se base essentiellement sur la **traduction du "Cairo Tutorial for Python Programmers", avec l'aimable autorisation de l'auteur original, Michael Urman** (Copyright © 2005-2008 Michael Urman). Le texte traduit, les images et codes python issus du site de M. Urman restent donc sous la licence GPL.
>
> Lorsque la première personne est utilisée dans le texte traduit, il faut comprendre “Michael Urman”. Si des informations sont ajoutées indépendantes de la traduction, ce sera clairement spécifié, et placé en fin de document.

Si vous êtes prêt à relever le défi, vous pouvez traduire les exemples vers le langage et l'environnement hôte de votre choix et n'utiliser que Cairo. Nis Martensen a gracieusement fait l'adaptation pour le langage C du présent document. Cette traduction C a été adoptée par le projet Cairo comme son tutoriel.

Remarque: Tout le code exemple a une dépendance sur cairo 1.2.0 ou ultérieur pour *cairo.SVGSurface*. En outre plusieurs exemples nécessitent *push_group()* et *pop_group()*, et les gradients radiaux nécessitent la version 1.4 pour un rendu correct. Vous pouvez contourner le premier cas, en changeant de nom de fichier *cairo.SVGSurface(filename + '.svg', width, height)* à *cairo.ImageSurface(cairo.FORMAT_ARGB32, width, height)*, mais en réalité vous devriez envisager de mettre à niveau Cairo.

## Principe de dessin de Cairo

Afin d'expliquer les opérations utilisées par Cairo, nous avons d'abord examiner de manière schématique la façon de dessiner de Cairo. Il y a seulement quelques concepts impliqués, qui sont ensuite appliqués à plusieurs reprises par les différentes méthodes. Je vais d'abord décrire les noms : destination, source, masque, chemin, et contexte. Ensuite je décrirai les verbes qui offrent les moyens de manipuler les noms et d'en tirer les graphiques que vous souhaitez créer. Voici le code à l'origine de la confection de tous les diagrammes, mais je vous conseille de ne pas le lire maintenant.

Si vous trouvez les descriptions ci-dessous trop clairsemée, Donn Ingle a créé des diagrammes

synoptiques en SVG qui tentent de lier le tout. Ils nécessitent Inkscape (ou un programme similaire) pour afficher, ainsi que deux polices spécifiques pour l'apparence correcte. Zoomez sur chaque «pages» au fur et à mesure de votre lecture. Comme Donn demande de télécharger et partager les diagrammes si on les trouve utiles, vous pourrez les télécharger en suivant ce lien.

## Noms

Les noms de Cairo sont un peu abstraits. Pour les rendre concrets, Michael Urman nous propose des diagrammes qui illustrent la façon dont ils interagissent. Les trois premiers noms sont les trois couches dans les schémas que vous voyez dans cette section. Le quatrième nom, le chemin, est placé sur la couche intermédiaire lorsque c'est pertinent. Le dernier nom, le contexte, n'est pas représenté.

### Destination

La destination est la surface sur laquelle vous dessinez. Elle peut être liée à une matrice de pixels, ou elle pourrait être liée à un fichier SVG ou PDF, ou autre chose. Cette surface recueille les éléments de votre graphique lorsque vous les appliquez, vous permettant de construire un travail complexe, comme une peinture sur une toile.

### Source

La source est la «peinture» avec laquelle vous allez travailler. Elle est présentée telle qu'elle est (un noir uni pour plusieurs exemples) ou de manière translucide pour montrer les couches inférieures. Contrairement à la vraie peinture, elle ne doit pas nécessairement être une seule couleur, cela peut être un motif de ou même une surface de destination préalablement créée. En outre, contrairement à la peinture réelle, la source peut contenir une information sur la transparence, le canal Alpha.

### Masque (mask)

Le masque est l'élément le plus important : il contrôle l'endroit où vous appliquez la source vers la destination. On le montrera comme une couche jaune avec des trous là où la source peut traverser. Lorsque vous appliquez un verbe de dessin, c'est comme si vous tamponnez la source sur la destination. Partout où le masque le permet, la source est copiée. Là où le masque l'interdit, rien ne passe.

### Chemin (path)

Le chemin est quelque part entre une partie du masque et une partie du contexte. Il sera illustré sous forme de fines lignes vertes sur la couche du masque. Il est manipulé par des verbes de chemin, puis

utilisé par les verbes de dessin.

#### Contexte (context)

Le contexte permet de suivre tout ce que les verbes affectent. Il suit une source, une destination, et un masque. Il suit également plusieurs variables auxiliaires comme la largeur de ligne et le style, le type et la taille de la police, et plus encore. Et surtout, il suit le chemin, qui est transformé en un masque par les verbes de dessin.

## Verbes

La raison pour laquelle on utilise Cairo dans un programme, c'est pour dessiner. Cairo fonctionne en interne avec une opération fondamentale de dessin : la *source* et le *masque* sont placés librement quelque part sur la *destination*. Ensuite les couches sont pressées ensemble et la peinture de la *source* est transférée vers la *destination* là où le *masque* le permet. Dans une certaine mesure, les cinq verbes suivants de dessin, ou les opérations, sont tous similaires. Ils diffèrent par la façon de construire le masque.

#### Tracé (stroke)

L'opération stroke() utilise un stylo virtuel le long du chemin (*path*). Elle permet le transfert de la *source* à travers le masque (*mask*) sur une ligne mince (ou épaisse) autour du chemin (*path*), en fonction de la largeur du stylo (line width), du style de ligne (dash style), et des extrémités de ligne (line caps).

Tutoriel Cairo : Diagrams (section #stroke)

```
cr.set_line_width(0.1)
cr.set_source_rgb(0, 0, 0)
cr.rectangle(0.25, 0.25, 0.5, 0.5)
cr.stroke()
```

#### Remplir (fill)



L'opération de remplissage ou fill() utilise plutôt le chemin (*path*), comme les lignes dans un livre de coloriage, et donne accès à la *source*, par l'intermédiaire du masque (*mask*) dont l'orifice est constitué par le chemin. Pour les chemins complexes (chemins avec de multiples sous-chemins fermés -comme un beignet- ou des chemins qui s'auto-intersectent) c'est influencé par la règle de remplissage. Notez que dans le cas du trait le transfert de la source le long du chemin se fait sur la moitié de l'épaisseur du trait de chaque côté de la trajectoire, tandis que le remplissage s'opère jusqu'aux bords définis par le chemin et pas au-delà.

Tutoriel Cairo : Diagrams (section #fill)

```
cr.set_source_rgb(0, 0, 0)
cr.rectangle(0.25, 0.25, 0.5, 0.5)
```

```
cr.fill()
```

### Afficher du texte / glyphes (Show Text / Glyphs)

L'opération //show_text()// forme le masque à partir d'un texte. On peut s'imaginer plus facilement *show_text()* comme un raccourci sur la création d'un chemin avec text_path() suivi du remplissage //fill()// pour son transfert. Soyez conscients que show_text() enregistre temporairement les glyphes, ce qui est d'autant plus efficace si vous travaillez avec beaucoup de texte.

Tutoriel Cairo : Diagrams (section #text)

```
cr.set_source_rgb(0.0, 0.0, 0.0)
cr.select_font_face("Georgia", cairo.FONT_SLANT_NORMAL,
cairo.FONT_WEIGHT_BOLD)
cr.set_font_size(1.2)
x_bearing, y_bearing, width, height = cr.text_extents("a")[: 4]
cr.move_to(0.5 - width / 2 - x_bearing, 0.5 - height / 2 - y_bearing)
cr.show_text("a")
```

### Peindre (paint)

L'opération peindre //paint()// utilise un masque qui transfère l'ensemble de la *source* vers la *destination*. Certaines personnes considèrent cela comme un masque infiniment grand, et d'autres considèrent cela comme une absence de masque; le résultat est le même. L'opération liée à paint_with_alpha() permet semblablement le transfert de la totalité de la source sur la destination, mais il ne transfère que le pourcentage spécifié de la couleur.

Tutoriel Cairo : Diagrams (section #paint)

```
cr.set_source_rgb(0.0, 0.0, 0.0)
cr.paint_with_alpha(0.5)
```

### Masque (mask)

Les opérations mask() et mask_surface() permettent le transfert selon la transparence/opacité d'un motif ou de la surface d'une seconde source. Lorsque le motif ou la surface est opaque, la source courante est transférée à la destination. Lorsque le motif ou la surface est transparente, rien n'est transféré.

Tutoriel Cairo : Diagrams (Section #mask)

```
self.linear = cairo.LinearGradient(0, 0, 1, 1)
self.linear.add_color_stop_rgb(0, 0, 0.3, 0.8)
self.linear.add_color_stop_rgb(1, 0, 0.8, 0.3)

self.radial = cairo.RadialGradient(0.5, 0.5, 0.25, 0.5, 0.5, 0.75)
self.radial.add_color_stop_rgba(0, 0, 0, 0, 1)
```

```
self.radial.add_color_stop_rgba(0.5, 0, 0, 0, 0)

cr.set_source(self.linear)
cr.mask(self.radial)
```

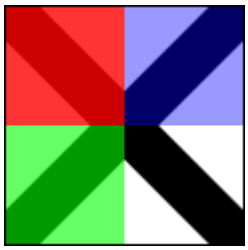
# Dessiner avec Cairo

Afin de créer une image que vous désirez, vous devez préparer le context pour chacun des verbes de dessin. Pour utiliser stroke() ou fill() vous avez d'abord besoin d'un chemin. Pour utiliser show_text(), vous devez positionner votre texte par son point d'insertion. Pour utiliser mask() vous avez besoin d'une deuxième source, pattern ou surface. Et pour utiliser n'importe laquelle de ces opérations, y compris paint(), vous avez besoin d'une source primaire.

## Préparation et Sélection d'une source

Il y a trois principaux types de sources dans Cairo: les couleurs, les gradients ou dégradés et les images. Les couleurs sont les plus simples, elles utilisent une teinte et une opacité uniformes pour la source entière. Vous pouvez les sélectionner sans aucune préparation avec set_source_rgb() et set_source_rgba(). L'utilisation de `set_source_rgb(r, g, b)` est équivalente à l'usage de `set_source_rgba( r, g, b, 1.0)`, et elle définit la couleur de votre source en utilisant une opacité complète.

Tutoriel Cairo: Drawing (section #rgba)



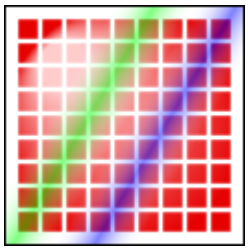
```
cr.set_source_rgb(0, 0, 0)
cr.move_to(0, 0)
cr.line_to(1, 1)
cr.move_to(1, 0)
cr.line_to(0, 1)
cr.set_line_width(0.2)
cr.stroke()

cr.rectangle(0, 0, 0.5, 0.5)
cr.set_source_rgba(1, 0, 0, 0.80)
cr.fill()

cr.rectangle(0, 0.5, 0.5, 0.5)
cr.set_source_rgba(0, 1, 0, 0.60)
cr.fill()
cr.rectangle(0.5, 0, 0.5, 0.5)
cr.set_source_rgba(0, 0, 1, 0.40)
cr.fill()
```

Des gradients décrivent une couleur variant progressivement en définissant un emplacement de début et un emplacement d'arrêt ainsi qu'une série de butées le long du chemin. Les gradients linéaires Linear gradients sont construits à partir de deux points qui définissent une direction sur laquelle on place les emplacements de début et d'arrêt. Des gradients radiaux (*radial gradients*) sont également construits à partir de deux points auxquels sont associés des rayons correspondant au cercle sur lequel on définit les emplacements de début et d'arrêt. Les butées (*stops*) sont ajoutés au dégradé avec add_color_stop_rgb() et add_color_stop_rgba() qui prennent une couleur comme `set_source_rgb*()`, ainsi qu'un décalage (*offset*) pour indiquer où il se trouve entre les emplacements de référence. Les couleurs entre les arrêts adjacents sont moyennées sur l'espace pour former un mélange fluide. Enfin, le comportement au-delà des emplacements de référence peut être contrôlé par set_extend().

Tutoriel Cairo: Drawing (section #gradient)



```
radiale = cairo.RadialGradient(0.25, 0.25, 0.1, 0.5, 0.5, 0.5)
radial.add_color_stop_rgb(0, 1.0, 0.8, 0.8)
radial.add_color_stop_rgb(1, 0.9, 0.0, 0.0)

for i in range(1, 10):
     for j in range(1, 10):
          cr.rectangle(i/10.0 - 0.04, j/10.0 - 0.04, 0.08, 0.08)
cr.set_source(radial)
cr.fill ()

linear = cairo.LinearGradient(0.25, 0.35, 0.75, 0.65)
linear.add_color_stop_rgba(0.00, 1, 1, 1, 0)
linear.add_color_stop_rgba(0.25, 0, 1, 0, 0.5)
linear.add_color_stop_rgba(0.50, 1, 1, 1, 0)
linear.add_color_stop_rgba(0.75, 0, 0, 1, 0.5)
linear.add_color_stop_rgba(1.00, 1, 1, 1, 0)

cr.rectangle(0.0, 0.0, 1, 1)
cr.set_source(linear)
cr.fill()
```

des images incluent deux surfaces chargées à partir des fichiers existants avec cairo.ImageSurface.create_from_png() et des surfaces créées à partir de Cairo comme destination préalable. Comme à partir de Cairo 1.2, la meilleure façon de faire et d'utiliser une destination préalable comme source se fait avec push_group() et soit avec pop_group() ou pop_group_to_source(). Utilisez `pop_group_to_source()` jusqu'à ce que vous sélectionnez une nouvelle source, et `pop_group()` lorsque vous voulez la sauvegarder de manière à pouvoir la sélectionner autant que désiré avec `set_source()`.

## Création d'un chemin

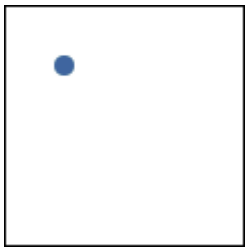
Cairo a toujours un chemin actif. Si vous appelez *stroke()* il dessinera le chemin avec vos paramètres de ligne. Si vous appelez *fill()*, il remplira l'intérieur du chemin. Mais comme le chemin est souvent vide, les deux appels ne se traduiront par aucun changement de votre destination. Pourquoi est-il si souvent vide ? D'une part, on commence dans cette configuration, mais plus important encore, après chaque stroke() ou fill(), le chemin est de nouveau vidé pour vous permettre de commencer à construire votre chemin suivant.

Que faire si vous voulez dessiner plusieurs choses avec le même chemin ? Par exemple, pour dessiner un rectangle rouge avec une bordure noire, vous voudriez remplir le chemin rectangulaire avec une source rouge, puis tracer le même chemin avec une source noire. Un chemin rectangulaire est facile à créer plusieurs fois, mais de nombreux chemins sont plus complexes.

Cairo supporte facilement la réutilisation des chemins en proposant des secondes variantes de ses opérations. Les deux dessineront la même chose, mais la seconde ne réinitialise pas le chemin. Pour tracer un trait, à côté de `stroke` il y a `stroke_preserve()`; pour le remplissage, `fill_preserve()` est l'alternative à `fill()`. Même la définition d'une région (clip) dispose d'une variante avec préservation.

A côté du choix de la sauvegarde du chemin, il y a seulement quelques opérations courantes :

### Mouvement

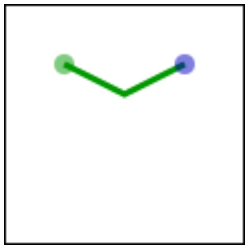


Cairo utilise un système de connecteurs entre points (connect-the-dots) lors de la création des chemins. Commencez au point 1, tracez une ligne vers 2, puis vers 3, et ainsi de suite. Lorsque vous démarrez un chemin, ou quand vous avez besoin de démarrer un nouveau sous-chemin, vous voulez qu'il le démarrer du point 1 sans que quelque chose y soit connecté. Pour cela, utilisez move_to(). Ceci définit le point de référence en cours sans créer un chemin de raccord à partir d'un point précédent. Il y a aussi une variante en coordonnées relatives, rel_move_to(), qui établit la nouvelle référence à une position spécifiée par une translation par rapport à la référence actuelle. Après avoir établi votre premier point de référence, utilisez les autres opérations de tracé de chemin pour mettre à la fois à jour le point de référence et s'y connecter d'une certaine manière.

Tutoriel Cairo: Drawing (section #moveto)

```
cr.move_to(0.25, 0.25)
```

### Lignes droites

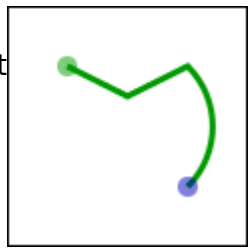


Que ce soit avec des coordonnées absolues line_to() (étendre le chemin de la référence à ce point), ou avec des coordonnées relatives rel_line_to() (étendre le chemin de la référence dans une direction donnée), le chemin de connexion sera une ligne droite. Le nouveau point de référence sera à l'autre extrémité de la ligne.

Tutoriel Cairo: Drawing (section #lineto)

```
cr.line_to (0.5, 0.375)
cr.rel_line_to (0.25, -0.125)
```

### Arcs

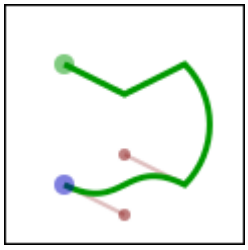


Les arcs sont des parties de l'extérieur d'un cercle. Contrairement aux lignes droites, le point que vous spécifiez n'est pas directement sur le chemin. C'est en fait le centre du cercle qui servira à définir l'arc, en spécifiant aussi le rayon, l'angle de départ et celui d'arrivée. Ces points sont reliés soit dans le sens horaire par arc() ou anti-horaire par arc_negative(). Si le point de référence précédent n'est pas sur la nouvelle courbe, une ligne droite est ajoutée à partir de lui jusqu'à l'endroit où l'arc commence. Le point de référence est ensuite mis à jour à l'endroit où l'arc se termine. Il n'y a que les versions absolues de ces deux opérations.

Tutoriel Cairo: Drawing (section #arc)

```
cr.arc (0.5, 0.5, 0.25 * sqrt(2), -0.25 * pi, 0.25 * pi)
```

### Courbes



Les courbes dans Cairo sont des courbes cubiques de Bézier. Elles commencent au point de référence en cours et sont tangentes en leurs extrémités à deux directions pointant vers deux autres points (sans passer par eux) pour se joindre à un troisième point (le point terminal) spécifié. Comme les lignes, il y a à la fois la version absolue curve_to() et la version relative rel_curve_to(). Notez que la variante relative précise l'ensemble des trois points par rapport au point de référence précédent, plutôt que de positionner chacun par rapport au point de contrôle précédent de la courbe.

Tutoriel Cairo: Drawing (section #curveto)

```
cr.rel_curve_to(-0.25, -0.125, -0.25, 0.125, -0.5, 0)
```

### Fermer le chemin

Cairo peut aussi fermer le chemin en traçant une ligne droite vers le début de l'actuel sous-chemin. Cette droite peut être utile pour la dernière arête d'un polygone, mais n'est pas directement utile pour des formes à base de courbes. Un chemin fermé est fondamentalement différent d'un chemin ouvert : c'est un chemin continu et il n'a pas de début ou de fin. Un chemin fermé n'a pas style de bout de ligne (*caps*) et il n'y a pas de possibilité d'en mettre (commande set_line_cap).

Tutoriel Cairo: Drawing (section #closepath)

```
cr.close_path()
```

**Texte**

Finalement, le texte peut être transformé en un chemin avec text_path(). Les chemins créés à partir de texte sont comme n'importe quel autre chemin, et supportent les opérations de tracé ou de remplissage. Ce chemin est placé ancrée au point de référence actuelle, nécessitant un *move_to()* vers l'emplacement de votre choix avant de transformer un texte en chemin. Cependant, il y a des problèmes de performance en utilisant cette fonction si vous travaillez avec beaucoup de texte; lorsque c'est possible, il est préférable d'utiliser l'opération *show_text()* au dessus de *text_path()* et *fill()*.

# Interprétation du texte

Pour utiliser du texte de manière efficace, vous avez besoin de savoir où il ira. Les méthodes font_extents() et text_extents() vont obtenir cette information. Si ce schéma est difficile à distinguer car trop petit, je vous conseille de prendre le code source et d'augmenter la taille jusqu'à 600. Il montre la relation entre le point de référence (point rouge); le point de référence suivant suggéré (point bleu); la boîte englobante (lignes en pointillés bleus), le déplacement de palier (ligne bleue) et les lignes de hauteur, d'ascension, de base, et de descente (en pointillés vert).

Le point de référence est toujours sur la ligne de base. La ligne de descente est en dessous, et reflète une boîte englobante approximative pour tous les caractères de la police. Toutefois, c'est un choix artistique destiné à indiquer l'alignement plutôt qu'une vraie boîte englobante. La même chose est vraie pour la ligne de montée au-dessus. Au dessus, on trouve la ligne de hauteur, l'espacement esthétiquement recommandé entre les lignes de base consécutives. Ce trois distances sont données comme des distances à la ligne de base, et devraient être positives en dépit de leurs directions différentes.

Le palier est le déplacement du point de référence dans le coin supérieur gauche de la boîte englobante. Il est souvent nul ou est une petite valeur positive pour le déplacement suivant x, mais il peut être négatif pour des lettre comme le **j** montré. C'est presque toujours une valeur négative pour le déplacement suivant y. La largeur et la hauteur décrivent alors la taille de la boîte englobante. L'avance vous positionne au point de référence proposé pour la lettre suivante. Notez que les rectangles englobant des blocs de texte consécutifs peuvent se chevaucher si le palier est négatif, ou que l'avance est plus petite que ce que la largeur suggère.

En plus du placement, vous devez également spécifier une police, son style et sa taille. On définit la police et le style avec select_font_face(), et la taille avec set_font_size(). Si vous avez besoin d'un contrôle encore plus fin, essayez d'utiliser cairo.FontOptions() avec get_font_options(), en les réglant avec set_font_options().

Lorsque vous travaillez dans GTK +, il y a aussi pangocairo.CairoContext. Michael Urman propose d'utiliser les fonctionnalités de caractères intégrées à Cairo, plus adaptée pour la manipulation de texte avec des restrictions de taille et des positionnements pointilleux complexes. Sinon, *pangocairo.CairoContext* peut-être plus adapté lorsque vous essayez de créer un widget texte qui ressemble à d'autre widget textes GTK+, ou qui a des contraintes complexes sur la police ou la disposition.

# Travailler avec des transformations

Les transformations ont trois utilisations principales. D'abord, elles vous permettent de mettre en place un système de coordonnées sur lequel il est facile de penser et de travailler, et d'avoir une sortie de taille quelconque. Deuxièmement, elles vous permettent de créer des fonctions auxiliaires qui travaillent dans ou autour d'un point *zéro* (0, 0), mais qui pourront être appliquée n'importe où dans l'image de sortie. Troisièmement, elles vous permettent de déformer l'image, de modifier un arc de cercle en un arc elliptique, ... Les transformations sont une façon de mettre en place une relation entre deux systèmes de coordonnées. Le système de coordonnées de l'espace du périphérique est attaché à la surface, et ne peut pas changer. Le système de coordonnées de l'espace utilisateur correspond à cet espace par défaut, mais peut être changé pour les raisons invoquées ci-dessus. Les fonctions auxiliaires user_to_device() et user_to_device_distance() indiquent ce que sont les coordonnées du périphérique pour une position ou distance en coordonnées de l'utilisateur. De la même manière device_to_user() et device_to_user_distance() vous donnent les coordonnées utilisateurs pour des positions ou distances en coordonnées périphériques. N'oubliez pas de fournir des positions à une variante non-distance des fonctions, et des déplacements relatifs par rapport ou d'autres distances aux variantes utilisant les distances.

la construction des diagrammes du présent document s'appuient sur l'ensemble de ces principes. Qu'on dessine en 120 x 120 ou 600 x 600 pixels, on utilise scale() pour considérer un espace de travail unitaire 1.0 x 1.0. Pour placer les résultats le long de la colonne de droite, comme dans la discussion du modèle de dessin de Cairo, on utilise translate(). Pour ajouter la vue en perspective des couches qui se chevauchent, j'ai (Michael Urman) mis en place une déformation arbitraire avec transform() sur une cairo.Matrix().

Pour comprendre vos transformations, il faut les lire de bas en haut, en les appliquant au point que vous dessinez. Pour comprendre quelle transformation créer, il faut penser au processus dans le sens inverse. Par exemple, si je veux que mon espace de travail 1.0 x 1.0 corresponde à 100 x 100 pixels au milieu d'une surface de 120 x 120 pixels, je peux mettre en place une des trois méthodes suivantes:

1. `cr.translate (10, 10); cr.scale (100, 100)`
2. `cr.scale (100, 100); cr.translate (0,1, 0,1)`
3. `cr.transform (cairo.Matrix (100, 0, 0, 100, 10, 10))`

Utilisez la première méthode lorsque c'est pertinent, car c'est souvent la plus lisible. Utilisez la troisième pour un contrôle supplémentaire indisponible avec les fonctions primaires *translate* et *scale*.

Soyez prudent lorsque vous essayez de dessiner des lignes lorsque vous êtes en mode de transformation. Même si vous réglez votre largeur de la ligne alors que le facteur d'échelle est de 1, le paramètre de largeur de ligne est toujours en coordonnées de l'utilisateur et n'est pas modifié en réglant le facteur d'échelle. Tant que vous opérez avec un facteur d'échelle, la largeur de votre ligne est multiplié par cette échelle. Pour spécifier une largeur d'une ligne en pixels, utilisez la fonction device_to_user_distance() pour transformer une distance (1, 1) de l'espace du périphérique en, par exemple, une distance (0,01, 0,01) de l'espace utilisateur. Notez que si votre transformation déforme l'image, il n'y a pas nécessairement moyen de produire une ligne avec une largeur uniforme.

# Que faire ensuite

Ceci clôture le tutoriel. Il ne couvre pas toutes les fonctions de Cairo, donc pour certaines fonctionnalités avancées mais moins répandues, vous aurez besoin de regarder ailleurs. Le code derrière les exemples utilise quelques techniques qui ne sont pas décrites, donc leur analyse peut être une bonne première étape. D'autres exemples sur cairographics.org vont dans différentes directions. Comme pour tout, il y a un grand écart entre la connaissance des règles de l'outil, et la capacité à bien l'utiliser. La dernière section de ce document donne quelques idées pour vous aider à franchir le cap.

# Trucs et astuces

Dans les sections précédentes, vous devriez avoir construit une solide connaissance des opérations que Cairo utilise pour créer des images. Dans cette section, j'ai (NDT : pour rappel, Michael Urman) rassemblé une petite poignée d'extraits que j'ai trouvé particulièrement utiles ou non évidents. Il peut y avoir d'autres façons pour mieux faire ces choses.

## Largeur de ligne

Lorsque vous travaillez sous une transformation d'échelle uniforme, vous ne pouvez pas juste utiliser des pixels pour la largeur de votre ligne. Toutefois, il est facile d'effectuer la transposition par le raccourci suivant:

```
cr.set_line_width (max (cr.device_to_user_distance (pixel_width,
pixel_width)))
```

Lorsque vous travaillez avec une échelle déformée, vous voudriez toujours avoir des largeurs de ligne qui sont uniformes dans l'espace périphérique. Pour cela, vous devez revenir à une échelle uniforme avant le tracé du chemin. Dans l'image, l'arc sur de gauche est tracé sous une déformation, tandis que l'arc de droite est tracé avec une échelle uniforme.

Caire Tutoriel: Tips and Tricks (article #deform)

```
cr.save()
cr.scale(0.5, 1)
cr.arc(0.5, 0.5, 0.40, 0, 2 * pi)
cr.stroke()

cr.translate(1, 0)
cr.arc(0.5, 0.5, 0.40, 0, 2 * pi)
cr.restore()
cr.stroke()
```

## Alignement du texte

Lorsque vous essayez de centrer le texte lettre par lettre à différents endroits, vous devez décider comment vous voulez effectuer ce centrage. Par exemple le code suivant en fait des lettres centrées individuellement, conduisant à des résultats médiocres lorsque vos lettres sont de tailles différentes. (Contrairement à la plupart des exemples, ici, on suppose un espace de travail 26 x 1.)

Cairo Tutoriel: Tips and Tricks (section #center)

```
for cx, letter in enumerate('abcdefghijklmnopqrstuvwxyz'):
    xbearing, ybearing, width, height, xadvance, yadvance =
(cr.text_extents(letter))
        cr.move_to(cx + 0.5 - xbearing - width / 2, 0.5 - ybearing - height
/ 2)
        cr.show_text(letter)
```

Le centrage vertical doit plutôt être basé sur la taille générale de la police, gardant ainsi votre ligne de base stable. Notez que le positionnement exact dépend maintenant des métriques prévues par la police elle-même, de sorte que les résultats ne sont pas nécessairement les mêmes entre polices différentes.

Cairo Tutoriel: Tips and Tricks (section #baseline)

```
fascent, fdescent, fheight, fxadvance, fyadvance = cr.font_extents()
for cx, letter in enumerate('abcdefghijklmnopqrstuvwxyz'):
    xbearing, ybearing, width, height, xadvance, yadvance =
(cr.text_extents(letter))
    cr.move_to(cx + 0.5 - xbearing - width / 2, 0.5 - fdescent + fheight /
2)
    cr.show_text (lettre)
```

## Références complémentaires, hors traduction

- http://zetcode.com/tutorials/cairographicstutorial/
- http://preshing.com/20110831/penrose-tiling-explained

From:
https://dvillers.umons.ac.be/wiki/ - **Didier Villers, UMONS - wiki**

Permanent link:
**https://dvillers.umons.ac.be/wiki/floss:python:cairo-tutoriel?rev=1330352819**

Last update: **2012/02/27 15:26**