Context Free Art en bref

I - Instructions de base

startshape
Indique quelle forme est utilisée comme point de départ d'une image générée. S'il y a plusieurs instructions startshape alors seule la première est prise en compte.

Exemple
```
startshape  maForme
shape maForme{
SQUARE[]
}
```

import
Importe le contenu du fichier spécifié après l'instruction import. Les guillemets sont optionnels pour les noms de fichiers sauf si l'on doit spécifier un répertoire, si le nom contient un espace ou ne finit pas en '.cdfg'. Le contenu peut aussi être inséré dans un espace de nom (namespace) différent.

Exemple
```
import@font alfreebet.cfdg
import "lib/alien eyes.cfdg"
```

Couleur de fond
Change la couleur de fond qui par défaut est blanche opaque. S'il y a plusieurs fonds spécifiés, seul le premier est pris en compte.
Exemple
```
CF::Background = [a -1] # fond transparent
```

Taille du fond
Fixe les dimensions de la toile plutôt qu'elle ne varie dynamiquement avec le dessin.
Exemple
```
CF::Size = [s 30]
CF::Size = [s 20 30] # fond rectangulaire
```

Aspect mosaïque
Active le rendu en mosaïque et défini la taille du pavage. S'il y a plusieurs types de pavage spécifiés, seul le premier est pris en compte. Le pavage peut-être carré, rectangulaire, incliné ou tourné tant que l'axe du pavage est parfaitement vertical ou horizontal.

Exemple
```
CF::Tile = [s 30]
CF::Tile = [s 20 30] # pavage rectangulaire
```

CF::Tile = [s 30 x 15 y 10]
Le décalage en x et y déplace le contenu par rapport au pavage. Cela permet à l'utilisateur de centrer une partie du dessin par rapport à un pavé.

II - Formes

Une forme peut être définie à partir d'autres formes, dans l'exemple ci-dessous, un cercle dans un carré. Il existe des formes de bases prédéfinies: SQUARE, CIRCLE, TRIANGLE ou FILL à partir desquelles on dessine des formes plus complexes. Dans une forme, ce sont les formes de bases qui sont évaluées en premier, les autres sont placées dans une liste pour être dessinées après.

Exemple
```
shape maForme{
SQUARE []
CIRCLE [b 1]
}
```

Comportements aléatoires
Une forme peut être définie par des règles (instruction rule). S'il y a plusieurs règles, la règle exécutée est choisie aléatoirement.

Exemple
```
shape maForme
rule { SQUARE[] }
rule { CIRCLE[] }
```

L'instruction rule peut-être suivie d'un nombre qui définit la probabilité relative pour que la règle correspondante soir exécutée. Ainsi, dans l'exemple ci-dessous, un cercle a 90,8% de chance d'être dessiné lors de l'exécution de la forme maForme.

Exemple
```
shape maForme
rule {SQUARE[] } # 1/11.01=9.1%
rule 10 {CIRCLE[] } # 10/11.01=90.8%
rule 0.01 {} # 0.01/11.01=0.09%
```

Dans l'exemple suivant, on a attribué directement des pourcentages aux règles sauf à la première dont la probabilité de ce produire sera 100% moins les pourcentages des autres règles.

```
shape maForme
rule {SQUARE[] } # 9.9% restant
rule 90% {CIRCLE[] } # 90%
rule 0.1% {} # 0.1%
```

III - Ajustement des formes

L'état d'une forme (position, couleur, etc) se définit par rapport à la forme précédemment dessinée. Ainsi, dans l'exemple suivant
```
shape maForme{
SQUARE []
CIRCLE [x 1]
}
```
un carré noir au centre de la toile est dessiné. Puis un cercle lui aussi noir décalé le long de l'axe x (horizontalement par défaut) de 100% (x 1) de la taille du carré précédent. La forme ci-dessous
```
shape maForme {
SQUARE []
CIRCLE [x 0.5 b 0.5]
}
```
correspond au dessin d'un carré noir suivi d'un cercle décalé horizontalement de 50% de la taille du carré (x 0.5) et 50% moins brillant (b 0.5) ce qui correspond à du gris.

Les dessins, sont repérés par 3 axes: x (par défaut horizontal), y (par défaut vertical) et z (qui détermine quelle forme est par-dessus l'autre).

Les variables d'ajustements de formes (x, size, etc) sont indiquées ci-dessous :
x val : Décalage val le long de l'axe x
x va1 val2 : Décalage val1 et val2 le long des axes x et y
x va1 val2 val3 : Décalage val1, val2, val3 le long des axes x, y et z
y val : Décalage val le long de l'axe y
z val : Décalage val le long de l'axe z
size val ou s val : Mise à l'échelle selon les axes x et y de la valeur val
size val1 val2 ou s val1 val2 : Mise à l'échelle selon les axes x et y des valeurs val1 et val2
size val1 val2 val3 ou s val1 val2 val3 : Mise à l'échelle selon les axes x, y, z des valeurs val1, val2, val3
rotate val ou r val : Rotation de val degrés
flip val ou f val : Réflexion par rapport à une ligne passant par l'origine du repère à val degrés
skew val 1 val2 : Cisaillement de val1 et val2 degrés par rapport aux axes x et y
time val1 val2 : Décalage de val1 et val2 du temps de naissance et de mort
timescale val1 : Changement d'échelle temporelle de val1

Il existe également une subtilité dans la notation par crochet. Ainsi dans SQUARE [mes ajustements], les ajustements sont exécutés dans l'ordre x, y, rotate, size, skew et flip et ce qui est dupliqué est supprimé. Alors que dans SQUARE [[mes ajustements]] les ajustements sont appliqués dans l'ordre dans lequel vous les écrivez.

IV - Ajustement des couleurs

hue *val* ou h *val*

Ajoute *val* à la teinte (hue), val allant de 0 à 360.
saturation *val* ou sat *val*
val prend une valeur entre 0 et 1. Si *val*<0, change la saturation de la couleur de *val*% vers 0. Si *val*>0, change la saturation de *val*% vers 1.
brightness *val* ou b *val*
val prend une valeur entre 0 et 1. Si *val*<0, change la brillance de la couleur de *val*% vers 0. Si *val*>0, change la brillance de *val*% vers 1.
alpha *val* a *val*
val prend une valeur entre 0 et 1. Si *val*<0, la forme devient plus transparente. Si *val*>0, la forme devient moins transparente.

Ainsi

```
shape maForme{
SQUARE []
CIRCLE [x 0.5 hue 240 sat 0.5 b 0.5 a -0.5]
}
```

dessine un carré et un cercle décalé de 50% (x 0.5) par rapport au carré, de couleur bleuté (hue 240 sat 0.5 b 0.5) et à moitié transparent (a -0.5)

V - Chemins

Il est possible de créer de nouvelles formes de base avec l'instruction path.
Exemple

```
startshape  carre
path carre{
   MOVETO( 0.5,  0.5)  #position initiale
   LINETO(-0.5,  0.5)   #trace une droite
   LINETO(-0.5, -0.5)   #trace une droite
   LINETO( 0.5, -0.5)   #trace une droite
   CLOSEPOLY()          #retourne en position (0.5,0.5) et referme le tracé
}
```

dessine un carré noir. Les commandes pour définir de nouvelles formes sont les suivantes
MOVETO(*xnum*, *ynum*): se positioner en un point du dessin
LINETO(*xnum*, *ynum*): tracer une droite depuis un point jusqu'à un autre en (*xnum*, *ynum*)
ARCTO(*xnum*, *ynum*, *xradius*, *yradius*, *angle*): tracé d'arc de cercle
ARCTO(*xnum*, *ynum*, *radius*): tracé d'arc de cercle
CURVETO(*xnum*, *ynum*, CF::Continuous): tracé d'arc de cercle
CURVETO(*xnum*, *ynum*, *xctrl1*, *yctrl1*) : tracé d'arc de cercle
CURVETO(*xnum*, *ynum*, *xctrl2*, *yctrl2*, CF::Continuous) : tracé d'arc de cercle
CURVETO(*xnum*, *ynum*, *xctrl1*, *yctrl1*, *xctrl2*, *yctrl2*) : tracé d'arc de cercle
CLOSEPOLY() / CLOSEPOLY(CF::Align): fermeture d'une courbe en partant du dernier point au point initial.

Toutes ces instructions en xxxTO ont leur équivalent en xxxREL ou la position du dernier point est alors relative à celle du premier point.

En ce qui concerne
STROKE [*adjustments*]
STROKE(*stroke flags*) [*adjustments*]
CF::IsoWidth, CF::ButtCap, etc.
FILL [*adjustments*]
FILL(*fill flags*) [*adjustments*]
CF::EvenOdd

VI - Expressions

Opérateurs mathématiques

( )	parenthèses
^	puissance
-, !	négation, booléen not
*, /	multiplication et division
+, -	addition et soustraction
--	retourne le résultat de la soustraction s'il est positif et 0 sinon
+-, ±, .., …	intervalles aléatoires
<, >, <=, ≤, >=, ≥, <>, ≠	comparaison
&&, ||, ^^	booléen

Fonctions mathématiques

sin, cos, tan, cot, en degrés
asin, acos, atan, acot, retourne des degrés
sinh, cosh, tanh, asinh, acosh, atanh
log, log10, sqrt, exp, abs, floor,
atan2(y, x), mod(x, y), divides(x, y), div(x, y),
infinity, factorial, sg,
rand_static(), rand_static(x), rand_static(x, y),
rand(), rand(x), rand(x, y),
randint(), randint(x), randint(x, y),
min(exp0, exp1, exp2, …),
max(exp0, exp1, exp2, …),
ftime, frame

VII - Boucles

Forme 1

loop *val* [*adjustements*]
{
  Instructions
}
Répète *val* fois les instructions *instructions* en ajustant à chaque fois (*adjustements*) des paramètres de formes et de couleurs.

Exemple
shape maForme{
loop 10 [x 0.1 hue 20]
{

```
  CIRCLE[sat 0.5 b 0.5]
}
}
```
Dessine des cercles décalés et de couleurs différentes qui se superposent.

Forme 2
```
loop var=val [adjustements]
{
  Instructions
}
```
Répète *val* fois les instructions *instructions* en ajustant à chaque fois des paramètres de formes et de couleurs. La valeur est ici attribuée à une variable *var* dont on peut se servir dans *instructions*.

Exemple
```
shape maForme{
loop i=10 [x 0.1 hue 20]
{
  CIRCLE[sat i b 0.5]
}
}
```
Cet exemple dessine des cercles de couleurs décalés qui se superposent en modifiant la saturation selon l'index i.

Forme 3
```
loop val [adjustements]
{
  Instruction1
}
finally
{
  Instruction2
}
```

ou
```
loop var=val [adjustements]
{
  instructions1
}
finally
{
  instructions2
}
```
Répète *instructions1* et finalement 1 fois *instructions2*

"val" peut s'écrire sous plusieurs formes. Cela peut être un nombre et la boucle se répète autant de fois que ce nombre. Cela peut être sous la forme "valeur de départ, valeur d'arrivée" ou encore sous la forme "valeur de départ, valeur d'arrivée, pas".

Dans ce dernier cas la boucle compte depuis la valeur de départ à la valeur d'arrivée de pas en pas. Par exemple "i=2,10,2" prend les valeurs i=2, 4, 6, 8, 10. S'il n'y a qu'une seule instruction, on n'est pas obligé de mettre des accolades. Ainsi

```
shape maForme{
loop 10 [x 0.1 hue 20]
CIRCLE[sat 0.5 b 0.5]
}
```

fonctionne aussi et répète 10 fois l'instruction CIRCLE en ajustant chaque fois sa position et sa couleur.

VIII - Conditions

Forme 1

```
if (condition) then
{
  instruction1
}
else
{
  instruction2
}
```

Si *condition* est respectée alors les *instruction1* sont exécutées sinon ce sont les *instruction2* . S'il n'y a pas de "sinon" on peut aussi écrire

```
if (condition) then
{
  instruction1
}
```

On peut aussi écrire
if (*cond, true_exp, false_exp*)
Evalue *cond* et retourne *true_exp* si cond ≠ 0 et *false_exp* si cond=0.

Forme 1

Une autre manière d'écrire des conditions est

```
switch (expression) {
case const-case-expression1: case-body1
case const-case-expression2: case-body2
case const-case-expression3: case-body3
case const-case-expression4: case-body4
…
else: else-body
}
```

switch évalue *expression* et compare le résultat au *const-case-expression*. Si l'un de ces cas correspond à ce qui a été évalué pour *expression*, il exécute l'instruction *case-*

body correspondante. S'il n'en trouve aucun, il exécute les instructions après else

Forme 3
select(*n, expr0, expr1, expr2, expr3,…*)
Evalue *n* et retourne *expr0* si *n*<1, *expr1* si 1<*n*<2, *expr2* si 2<*n*<3, etc. Les *expr* peuvent être des expressions numériques, des ajustements de forme, des spécifications de forme, etc. Ils doivent tous être du même type.

Forme 4
let(*var1=expr1; var2=expr2; … ; expression*)
Evalue les *expr* et les associe au *var*. Alors *expression* est évalué dans le contexte des variables liées et retourne sa valeur. Une fois var1 calculé, il peut être utilisé, par exemple, dans *expr2*.

IX - Transform

transform prend une liste d'ajustements qu'il applique à une forme définie dans le corps de la fonction transform

Exemple

```
startshape  box

path box {
MOVETO( 0.5,  0.5)
LINETO(-0.5,  0.5)
LINETO(-0.5, -0.5)
LINETO( 0.5, -0.5)
CLOSEPOLY()
transform [s 0.5 x 0.1] {
  MOVETO(0.25,  sqrt(3)/4)
  loop 5 [r -60]
  LINETO(0.5, 0)
  CLOSEPOLY()
  }
}
```

X - Définition des variables

Une variable s'écrit comme

maVariable = maValeur
où maValeur peut être un nombre

```
shape maForme{
maVariable = 0.5
SQUARE[sat 0.5 b maVariable]
}
```

une forme

```
shape maForme{
maVariable = SQUARE
maVariable[sat 0.5 b 0.5]
}
```

un ajustement

```
shape maForme{
maVariable = [[sat 0.5 b 0.5]]
SQUARE[transform maVariable]
}
```

Notez dans ce dernier cas, l'utilisation de l'instruction transform. La valeur d'une variable ne peut pas être changée une fois qu'elle a été définie.

XI - Expression

Variables aléatoires
x .. y ou x … y retourne un nombre aléatoire entre [x,y)
x +- y or x ± y retourne un nombre aléatoire entre [x-y,x+y)

Symboles unicode
≤, ≥, ≠, ∞ (infini), π (3.1415926535)

Fonctions entières
div(*x, y*) retourne la division entière de x par y
divides(*x, y*) retourne 1 si x est divisible par y et 0 sinon
factorial(*n*) vaux 1x2x3…xn
floor(*x*) arrondit x à l'entier le plus petit
isNatural(*x*) retourne 1 si x est un entier naturel
sg(*x*) retourne 0 si x=0 ou 1 sinon

Fonctions binaires
bitnot(*x*) inverse binaire de x
bitand(*x, y*) fonction logique AND
bitor(*x, y*) fonction logique OR
bitxor(*x, y*) fonction logique XOR
bitleft(*x, y*) décalage vers la gauche de x de y bits
bitright(*x, y*) décalage vers la droite de x de y bits

Fonctions diverses
infinity() est l'infini ∞
infinity(x) renvoie ∞ si x ≥0 ou -∞ si x<0
max(*x0, x1, x2, …*) retourne le plus grand élément de la liste
min(*x0, x1, x2, …*) retourne le plus petit élément de la liste

Fonctions aléatoires

rand_static() retourne un nombre statique (lorsque le fichier est compilé) aléatoire dans l'intervalle [0,1) avec une distribution uniforme

rand_static(x) retourne un nombre statique (lorsque le fichier est compilé) aléatoire dans l'intervalle [0,x) si x>0 ou [x,0) si x < 0, avec une distribution uniforme

rand()/rand(*x*)/rand(*x*,*y*) retourne un nombre aléatoire dans les mêmes intervalles que les rand_static

rand::normal(*mean, stddev*): retourne un nombre aléatoire selon une distribution normale (ou gausienne)

rand::lognormal(*mean, stddev*): retourne un nombre aléatoire selon une distribution aléatoire log-normale

rand:: exponential(*rate*): retourne un nombre aléatoire selon une distribution de probabilité exponentielle.

rand::gamma(*alpha_shape, beta_scale*): retourne un nombre aléatoire selon la distribution aléatoire gamma.

rand::weibull(*alpha_shape, beta_scale*): retourne un nombre aléatoire selon la distribution aléatoire weibull.

rand::extremeV(*location, scale*): retourne un nombre aléatoire selon la distribution aléatoire de la valeur extrême.

rand::chisquared(*degree_freedom*): retourne un nombre aléatoire selon la distribution aléatoire en chi carré.

rand::cauchy(*location, scale*): retourne un nombre aléatoire selon la distribution aléatoire de Cauchy.

rand::fisherF(*m_degree_freedom, n_degree_freedom*): retourne un nombre aléatoire selon la distribution de Fishers F-.

rand::studentT(*degree_freedom*): retourne un nombre aléatoire selon la T-distribution de Student.

randint()/randint(*x*)/randint(*x, y*):: retourne un entier aléatoire dans le même intervalle que les rand_static.

randint::bernoulli(*probability*): retourne un booléen aléatoire avec une probabilité spécifiée d'être vraie.

randint::binomial(*trials, probability*): retourne un entier positif selon une distribution binomiale.

randint::negbinomial(*trial_failures, probability*): retourne un entier positif selon une distribution binomiale négative.

randint::geometric(*probability*): retourne un entier positif qui représente le nombre d'essai oui/non nécessaire pour obtenir un unique succès.

randint::poisson(*mean*): retourne un entier positif selon une distribution de Poisson.

randint::discrete(*weight_0, weight_1, … , weight_n*): retourne un nombre aléatoire dans l'intervalle [0,n] ou chaque valeur i se trouve avec une probabilité selon le poids *weight_i*.