

PAGE DE TITRE

Jordan Chavanne

Discipline : Informatique

Maître responsable : Didier Müller

Titre :

Étude du logiciel Context Free Art

Lycée cantonal de Porrentruy

Samedi 4 février 2012

Sommaire :

1. Introduction.....	Page 4
2. Mode d'emploi du programme	Page 4
2.1 Télécharger et installer Context Free Art	Page 4
2.2 Fonctionnalités de base du programme.....	Page 5
3. Mode d'emploi de la grammaire	Page 9
3.1 Formes.....	Page 9
3.2 Règle	Page 13
3.3 Fractale.....	Page 14
3.4 Hasard	Page 21
3.5 Chemin	Page 28
3.6 Couleurs	Page 35
3.7 Projet finale.....	Page 39
4. Application du logiciel	Page 46
5. Conclusion.....	Page 47
6. Médiagraphie.....	Page 48
7. Déclaration	Page 48
8. Table des matières	Page 49

1. Introduction

Context Free Art est un programme gratuit qui permet de créer des images grâce à du code. On appelle ce code une *grammaire*. Cette *grammaire* fonctionne un peu comme les codes des langages de programmation. Cela signifie que l'on retrouve des termes et des notions qui appartiennent à la programmation en général, par exemple les concepts de règle et d'argument. Tout cela est évidemment expliqué au cours de ce mode d'emploi.

Context Free Art est disponible pour les systèmes d'exploitation Windows, Macintosh, Linux et ses variantes.

Bien qu'une version beta 3 soit disponible sur le site officiel du logiciel, le mode d'emploi qui suit se base sur la version 2.2.2.

Le but de ce travail de maturité est de créer un mode d'emploi pour le logiciel Context Free Art. Ce mode d'emploi se divise en plusieurs parties. Le sommaire de la page précédente indique les points principaux de ce document, tandis que la table des matières présente à la fin fait référence à tous les chapitres et sous-chapitres de ce dossier. Il est à noter que ce travail se divise en deux grandes parties : il y a d'abord un mode d'emploi du programme en lui-même, puis une seconde partie où il ne sera question que de la grammaire utilisée par le programme.

L'utilisation du programme étant plutôt aisée, j'ai préféré ne pas passer trop de temps sur les fonctions basiques du programme mais plutôt mettre l'accent sur les outils selon moi utiles, intéressants et originaux mis en avant par le programme.

2. Mode d'emploi du programme

Voici un court mode d'emploi du programme. Il ne s'agit ici non pas d'expliquer comment fonctionne le code utilisé par le programme mais simplement comment se servir des différentes fonctionnalités proposées par ce dernier.

2.1 Télécharger et installer Context Free Art

Rendez-vous sur le site officiel du logiciel, <http://www.contextfreeart.org/>, puis allez dans l'onglet « *Download* ». Une fois arrivé sur la page, plusieurs liens de téléchargement sont disponibles, en fonction de la version du programme et du système d'exploitation spécifié.

Étant donné que le mode d'emploi s'est fondé sur la base de la version 2.2.2, c'est cette version-là que je recommande d'utiliser. En effet, les explications fournies et les méthodes utilisées dans ce mode d'emploi peuvent ne pas fonctionner dans les autres versions.

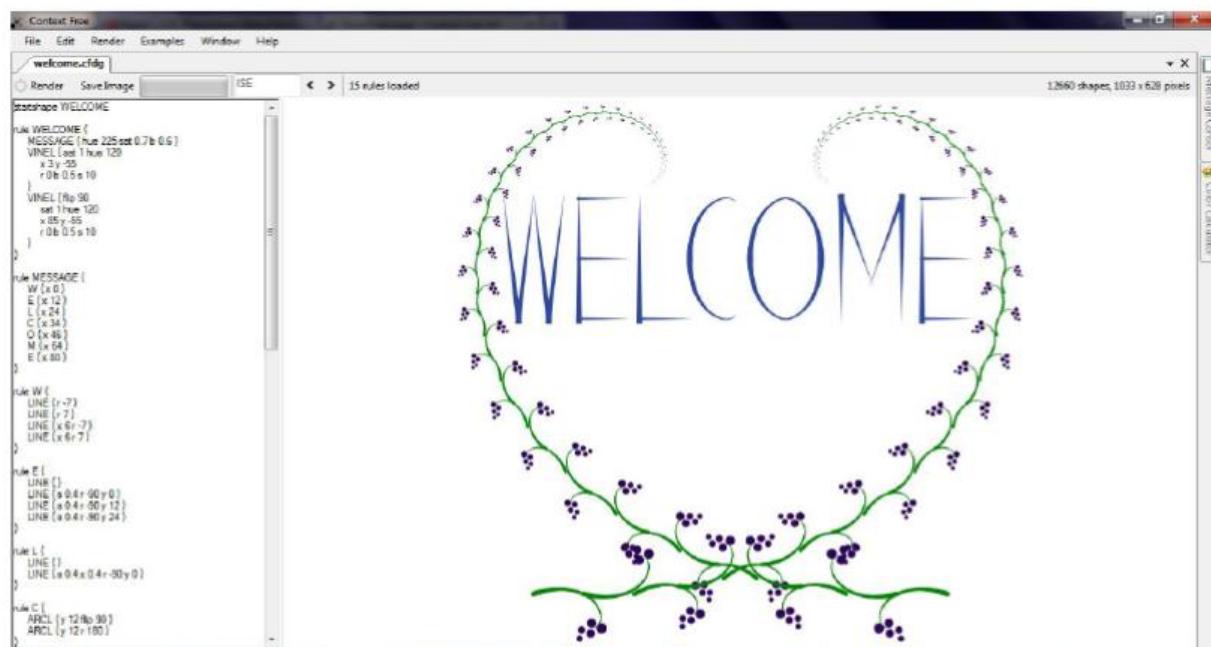
Une fois que vous avez téléchargé le programme, il va falloir l'installer.

Si vous avez téléchargé le programme sous forme de fichier *exécutable*, lancez-le puis suivez les instructions qui apparaissent à l'écran.

Si vous avez téléchargé le programme sous forme de dossier *zip*, vous devez extraire les fichiers contenus dans ce dossier et les copier dans un autre.

2.2 Fonctionnalités de base du programme

Une fois le programme téléchargé et installé, lancez-le. Une fenêtre devrait apparaître, semblable à celle-ci :



Première observation : tout est en anglais. Ensuite, examinez les divers onglets qui se trouvent en haut à gauche de la fenêtre. Ils sont au nombre de 6. Ce sont des onglets similaires à ceux que l'on trouve dans d'autres programmes.

Tout à gauche se trouve un espace réservé à l'écriture du code du programme. C'est là que l'on tape toutes les lignes de codes nécessaires à la création d'une image.

Il y a déjà du code lorsque vous ouvrez le programme ; c'est normal : le logiciel charge automatiquement une image par défaut, en affichant le code correspondant dans la partie gauche de la fenêtre.

Remarquez que la plus grande partie occupée par la fenêtre est celle où se trouve l'image obtenu grâce au code.

Cette partie du mode d'emploi se divise maintenant en plusieurs sous-parties : la première concerne les onglets, puis la deuxième est réservée à la console du programme et à l'outil « *Color calculator* » et finalement la dernière traite des boutons et des sous-onglets du programme.

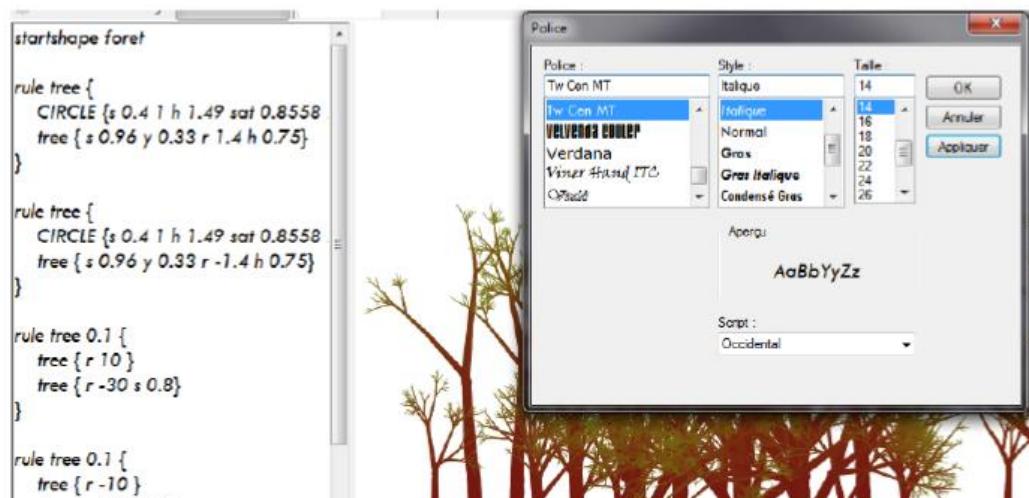
2.2.1 Les onglets

Au nombre de 6, ils permettent d'effectuer diverses actions, toutefois assez basiques. C'est pourquoi je vais uniquement survoler la liste des menus sans donner d'explications exhaustives.

Tout d'abord l'onglet « *file* », soit « *fichier* » en français. Il permet de créer un nouveau fichier, d'en enregistrer un, etc.

Le menu contient aussi la fenêtre des préférences. C'est là que vous pouvez choisir si vous voulez que le programme dessine l'image petit à petit en cours de *render* (le *render* est un rendu sous forme d'image du code écrit dans la partie prévue à cette effet) ou non, ou encore vous pouvez choisir ce que fait le programme une fois lancé (Par défaut, il charge automatiquement une image de bienvenue).

L'onglet « *edit* », l'équivalent d' « édition » en français, vous permet d'effectuer des actions basiques comme annuler une action ou la refaire, en revenant en arrière ou en avant. Dans cet onglet se trouvent également deux petits outils très utiles ; il s'agit de « *set font* » et de « *Find/Replace* ».



« *set font* » permet, comme on le voit sur l'image ci-dessus, de changer le type, le style et la taille de la police avec laquelle le code est écrit.

Quant à « *Find/Replace* », le plus utile à mon avis, il vous permet de rechercher une séquence de caractère de votre choix et de la remplacer par une autre si vous le desirez. Observez l'image suivante :



Lorsque vous cliquer sur « *Find/Replace* », une barre semblable à celle présente sur l'image du haut apparaît vers le bas de la fenêtre. Si vous observez l'image, en cliquant sur « *find* », le programme va s'arrêter à chaque fois qu'il trouve une séquence de caractère similaire dans le code. À chaque fois, on pourra choisir de la remplacer ou non en cliquant sur le bouton « *replace* ». Pour remplacer un mot puis directement rechercher le suivant, il faut utiliser le bouton « *replace & find* ».

L'onglet « *render* », ce qui signifie « *rendre* », vous permet de faire un rendu du code qui est écrit dans l'espace prévue à cet effet. En bref, il va créer l'image qui correspond au code qui est écrit. Il permet aussi de sauvegarder une image sur le disque dur ou encore de faire un rendu d'une image dont la taille peut être définie. L'onglet « *examples* » propose une liste d'images d'exemples dont le code a déjà été réalisé.

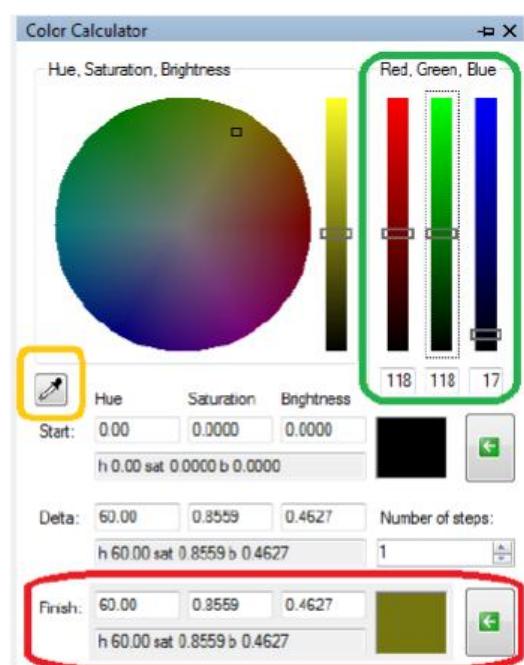
L'onglet « *window* », dit « *fenêtre* » en français, s'intéresse uniquement aux deux outils dont on va parler dans la deuxième sous-partie.

Et pour finir, l'onglet « *help* », dont la traduction française est « *aide* ». Rien de spéciale à signaler ici.

2.2.2 La console du programme et l'outil « *color calculator* »

Ces deux outils se trouvent à la droite de la fenêtre du programme (voir l'image ci-contre). S'ils n'y sont pas, il vous faut alors aller dans l'onglet « *window* » pour les activer, en cliquant sur leur nom respectif.

Le premier outil, qui est en fait la console du programme, est une sorte de messagerie dans laquelle presque tout ce que fait le programme est écrit. Mis à part cela, cette console n'a pas d'utilité particulière qui puisse vous servir.



L'autre outil, bien plus intéressant, est le calculateur de couleur. Nous l'utiliserons beaucoup lorsque nous aurons besoin de couleur, mais pour le moment, nous n'en avons pas d'utilité. Je vous invite donc à juste jeter un œil à cette section et à y revenir en temps voulu, lorsque nous aborderons la couleur dans la deuxième partie du mode d'emploi.

L'image ci-contre montre sous quelle forme se trouve cet outil.



Il vous servira à trouver les valeurs de hue, de saturation et de luminosité pour obtenir une certaine couleur.

Les niveaux de rouge, vert et bleu sont encadrés en vert sur l'image précédente. Les curseurs de chaque colonne peuvent être déplacés afin de modifier la couleur.

Le sélecteur de couleur est encadré en jaune. Si vous cliquez dessus, l'outil s'enclenche et attend que vous cliquez une deuxième fois, sur une image dont la couleur vous intéresse. Une fois ceci fait, l'ordinateur trouve automatiquement le code correspondant à cette couleur dans le calculateur de couleur.

À l'intérieur de ce qui est encadré en rouge se trouvent les valeurs finales qui correspondent à la couleur désirée, que vous placerez dans votre code.

2.2.3 Les boutons et les sous-onglets

Nous allons maintenant nous attaquer aux boutons et aux sous-onglets. On remarque rapidement les boutons « *render* » et « *save image* », qui se trouvent au-dessus de l'espace réservé au code.

Les sous-onglets se trouvent quant à eux juste au-dessus des boutons. Chaque fois que vous ouvrez ou créez un nouveau fichier, un nouvel onglet apparaît. Cela fonctionne un peu comme les onglets de certains navigateurs web.

Voici une observation plus détaillée à l'aide d'une image :



1. Bouton « *render* » : permet d'effectuer un rendu du code. Lorsque l'ordinateur est en train de faire un « *render* », ce bouton change de nom et d'effet et il permet alors d'arrêter le processus de *rendering* en cours.
2. Bouton « *save image* » : permet de sauvegarder l'image obtenue.
3. Sous-onglet : Ici, comme nous n'avons qu'un seul fichier d'ouvert, il n'y a qu'un seul sous-onglet. Mais si nous créons un nouveau fichier, son sous-onglet apparaîtra à côté de celui déjà présent.
4. Barre de progression du *rendering* : une fois que la barre est complètement verte, cela veut dire que le processus de *rendering* est terminé.
5. Flèche et croix : la flèche permet de naviguer à travers les onglets, la croix de supprimer l'un d'.

3. Mode d'emploi de la grammaire

Cette partie ne concerne que la grammaire dont il est question. Pour rappel, on appelle grammaire le code utilisé par le logiciel Context Free Art.

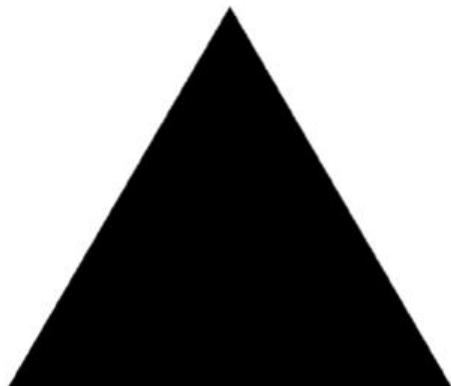
3.1 Les formes

3.1.1 Création de formes de base

Le programme commence toujours par la règle définie par *startshape*. Une *règle* est une séquence d'instruction que l'ordinateur va effectuer. Dans l'exemple suivant, nous n'avons qu'une seule règle et nous pouvons la nommer comme nous voulons.

```
startshape dessin  
  
rule dessin {  
    TRIANGLE {}  
}
```

Donne :



D'autres formes que le triangle sont disponibles, comme le carré (*SQUARE{}*) ou le cercle (*CIRCLE{}*). Dans l'exemple suivant, le code utilisé précédemment a été repris et modifié pour que l'ordinateur fasse un triangle puis un carré éloigné du triangle. Pour ce faire, l'instruction *SQUARE{x 5}* a été ajoutée ; elle permet de créer un carré qui est décalé de 5 sur l'axe des x (par rapport à la dernière position où l'on se trouvait).

```
startshape dessin  
  
rule dessin{  
    TRIANGLE{}  
    SQUARE{x 5}  
}
```

Donne :



3.1.2 Répétition d'une même forme

Si maintenant nous essayons ce code :

```
startshape dessin  
  
rule dessin{  
    CIRCLE()  
    dessin{x 2 s 0.8}  
}
```

Nous obtiendrons :



Comment se fait-il ?

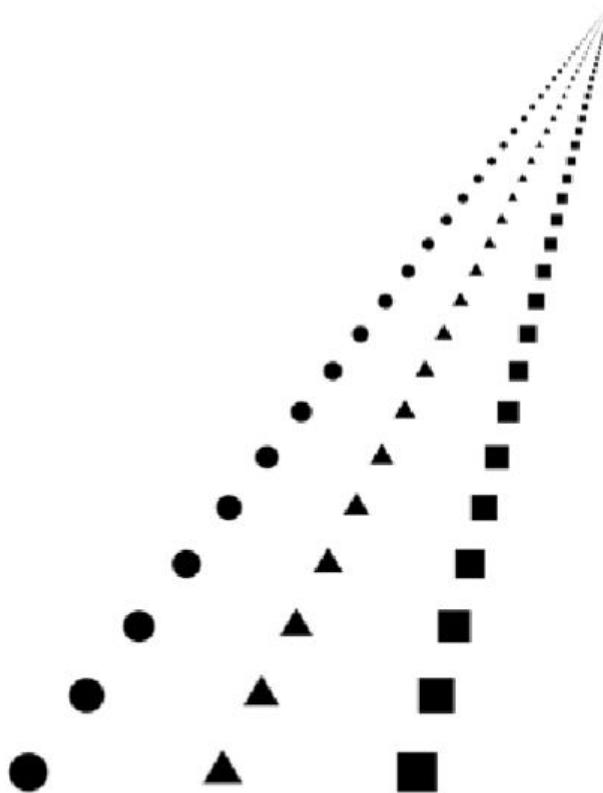
L'ordinateur va lancer une première fois la fonction *dessin*.

À la fin de celle-ci, il lit une instruction qui lui demande d'exécuter la règle *dessin*, avec deux *arguments*: *x 2* et *s 0.8*. En fait, le premier argument décale les prochaines figures de l'axe des *x* de 2. Si nous avions mis *y 2*, les figures suivantes auraient été décalées de 2 sur l'axe des *y*. Quant à l'autre argument, il modifie la taille du cercle en la multipliant par une valeur. Ici, la valeur étant 0.8, le cercle va rétrécir. *s* est en fait le raccourci de *size*. Lorsque vous voulez utiliser cet argument, les deux formes sont valables, il n'y a aucune différence entre les deux si ce ne que l'une est écrite sous forme de raccourci.

Et ainsi de suite : l'ordinateur va refaire un cercle toujours en le décalant et en le redimensionnant, car nous avons indiqué des arguments entre les accolades.

Notre règle s'exécute en boucle puisque, dans la ligne *dessin{x 2 s 0.8}*, la fonction appelle la fonction *dessin* (« appelle » signifie « exécute », avec les arguments fournis entre accolades s'il y en a). En fait, la fonction s'appelle elle-même. C'est ce qu'on désigne par fonction récursive. C'est une boucle infinie puisque le programme ne s'arrête jamais de décaler puis de rétrécir le cercle. Cependant, le programme s'arrête automatiquement de dessiner dès que les formes ou les dessins qu'il crée deviennent vraiment très petits. Si l'ordinateur effectue tout de même des instructions à l'infini sur une image sans en modifier la taille, vous devez vous-même arrêter le programme en appuyant sur le bouton « *stop* », qui est en fait le bouton « *render* » (Rappelez-vous que ce dernier change de nom une fois le processus de *rendering* lancé !).

Voici encore un autre exemple où l'ordinateur va rétrécir une forme jusqu'à ce qu'elle devienne trop petite pour la voir à l'œil nu ; il va dès lors cesser de créer de nouvelles formes.



```
startshape dessin
rule dessin {
    TRIANGLE{}
    SQUARE{x 5}
    CIRCLE{x -5}
    dessin {x 1 y 2 s 0.9}
}
```

Donne, ci-contre :

On crée d'abord un triangle, puis un carré décalé de 5 sur l'axe des x et finalement nous ajoutons un cercle que nous décalons de -5 sur l'axe des x (donc de 5 dans le sens opposé). Puis à la fin de la règle on appelle la règle *dessin* (la règle s'appelle elle-même), en donnant comme arguments *x 1*, *y 2* et *s 0.9*. *x 1* décale chaque fois la nouvelle forme que l'ordinateur crée de 1 vers l'axe des x, *y 2* fait de même en décalant les formes de 2 vers l'axe des y et *s 0.9* multiplie la taille à chaque fois par 0.9 (et donc elles rétrécissent).

Maintenant, observez le code suivant :

```
startshape dessin
rule dessin{
    CIRCLE{}
    dessin{x -0.1 s 0.99 r 1}
}
```

Donne :



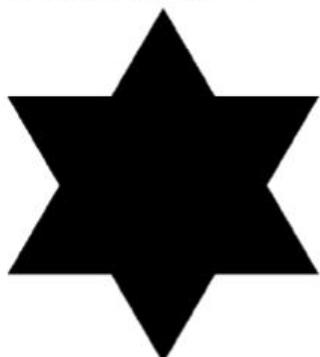
Il s'agit encore une fois d'une boucle : dans la règle *dessin*, l'instruction *dessin{x-0.1 s 0.99 r 1}* ordonne à l'ordinateur d'exécuter la règle *dessin* en décalant le cercle de -0.1 et en le rétrécissant à chaque fois. Puis, à l'aide de l'argument *r 1* on fait faire au cercle créé un angle de 1 degré.

3.1.3 Étoiles

Voici une petite astuce pour pouvoir créer des étoiles.
Si vous essayer les quelques lignes suivantes :

```
startshape faire_etoile
rule faire_etoile{
    TRIANGLE{}
    TRIANGLE{r 60}
}
```

Vous obtiendrez :



Il s'agit d'une astuce toute simple : on crée d'abord un triangle. Ensuite, on crée un deuxième triangle et on effectue une rotation de 60 degré sur ce triangle.

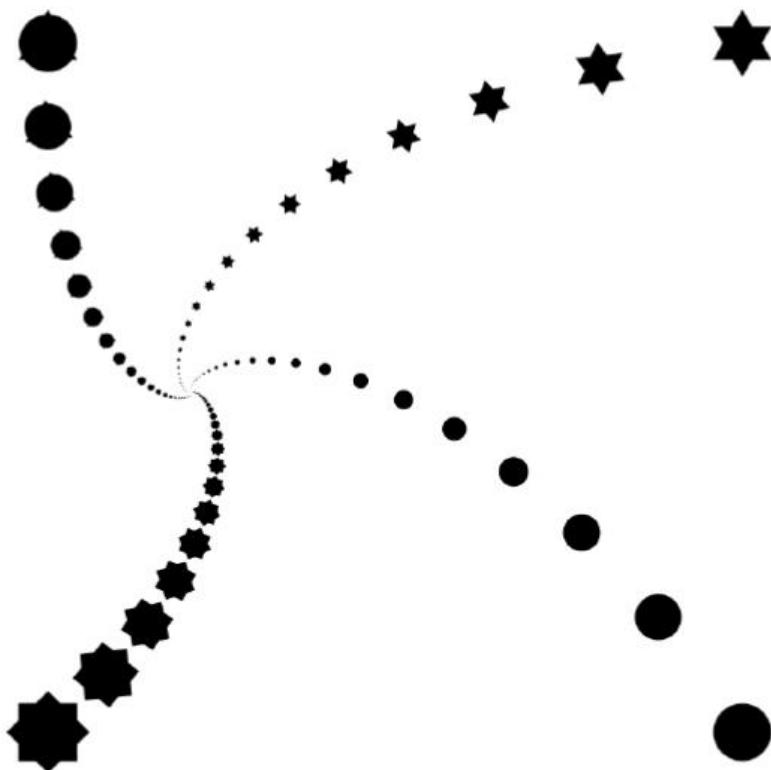
Vous pourrez dès lors utiliser cette astuce afin de crée des étoiles, mais n'hésitez pas à changer les types de formes ou l'angle pour obtenir de nouvelles formes.

3.1.4 En résumé des trois derniers points :

Le code ci-dessous utilise tout ce que nous avons appris jusqu'à maintenant. Observez bien de quelle manière nous avons créé des formes « spéciales ».

```
startshape faire_etoile
rule faire_etoile{
    SQUARE{}
    SQUARE{r 45}
    CIRCLE{x 12}
    TRIANGLE{x 12 y 12}
    TRIANGLE{x 12 y 12 r 60}
    CIRCLE{y 12}
    TRIANGLE{y 12}
    faire_etoile{x 1 y 1 r 6 s 0.8}
}
```

Donne (page suivante) :



3.2 Système de règles

3.2.1 Pousser un peu plus loin le système de règle

Jusqu'à maintenant, nous n'avons utilisé qu'une seule et unique règle. Mais nous pouvons évidemment employer autant de règles que nous voulons.

Pour rappel, une règle est une série d'instructions bien définies. On peut « appeler » une règle, c'est-à-dire qu'on va lire et exécuter les instructions qui y sont présentes. Lorsqu'on appelle une règle, on peut transmettre des arguments, qui sont des valeurs spéciales, permettant de changer par exemple la taille d'une figure en la rétrécissant toujours plus.

Essayons maintenant d'utiliser plusieurs règles. Observez le code de la page suivante. Nous avons repris celui que nous avions utilisé pour la dernière image et nous avons ajouté une règle au début. Nous avons également changé la règle définie après `startshape`. Pour rappel, le nom de la règle que vous écrivez à la suite de `startshape` sera la règle que le programme exécutera en premier.

```

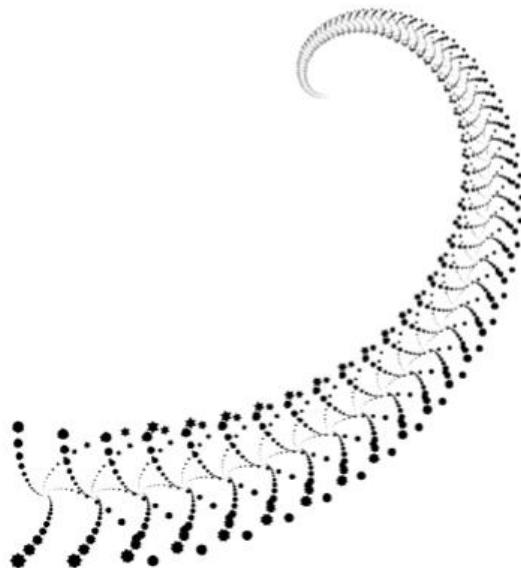
startshape decaler_le_tout

rule decaler_le_tout{
    faire_etoile{}
    decaler_le_tout{r 5 s 0.95 x 5}
}

rule faire_etoile{
    SQUARE{}
    SQUARE{r 45}
    CIRCLE{x 12}
    TRIANGLE{x 12 y 12}
    TRIANGLE{x 12 y 12 r 60}
    CIRCLE{y 12}
    TRIANGLE{y 12}
    faire_etoile{x 1 y 1 r 6 s 0.8}
}

```

Donne :



Premièrement, vous pouvez remarquer que la règle *faire_étoile* qui était celle par laquelle nous commençons dans notre code précédent est exactement la même dans ce nouveau code. Elle n'est cependant plus la règle par laquelle l'ordinateur débute.

Nous avons ajouté une nouvelle règle au début de notre programme ; *decaler_le_tout* (notez qu'on peut sans autre insérer des « _ » dans le nom d'une règle).

C'est par cette règle que le programme commence. Mais que fait-elle exactement ? *Faire_etoile{} :* cette instruction appelle la règle *faire_etoile* (sans argument particulier). On va donc créer la figure que nous avons construite lors de l'exercice précédent (voir plus haut).

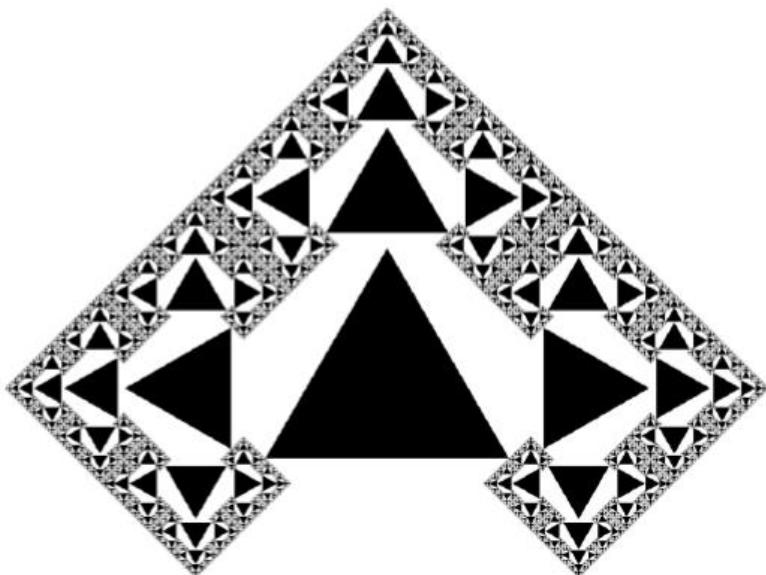
Decaler_le_tout{r 5 s 0.95 x 5} : Cette instruction appelle la fonction *decaler_le_tout* (la fonction s'appelle donc elle-même). Elle va de nouveau exécuter l'instruction *faire_etoile*, mais avec des arguments (puisque nous avons indiqué des arguments en appelant la règle elle-même). À chaque fois, la figure obtenue grâce à la règle *faire_etoile* subira :

- Une rotation de 5 degré
- Un décalage sur l'axe des x de 5
- Une multiplication de taille de 0.95, ce qui revient à une réduction de la figure.

3.3 Fractale

3.3.1 Triangle de Sierpiński et son dérivé

Context Free Art est un logiciel utilisant énormément la récursivité, ce qui permet de réaliser des fractales. Voici sans plus attendre un exemple de fractale réalisée sous Context Free Art.



Observez le code qui nous permet d'obtenir une telle figure¹ :

```
startshape fractal

rule fractal {
    TRIANGLE {}
    fractal { y  0.75 s 0.5}
    fractal { x  0.75 s 0.5 r -90}
    fractal { x -0.75 s 0.5 r 90}
}
```

Comment l'obtient-on ?

En fait, afin de réaliser cette fractale, on utilise une règle qui crée un simple triangle et on appelle la règle elle-même avec différents arguments.

Dans la règle *fractal*, on crée d'abord un triangle. Puis on appelle la règle *fractal* avec divers arguments. La règle *fractal* est en fait la règle dans laquelle on se trouve à ce moment-là, c'est-à-dire qu'elle est la règle qui crée un simple triangle. Le fait d'appeler la règle *fractal* avec des arguments va créer un triangle d'une autre manière. La première fois qu'on appelle la règle *fractal*, on le fait avec les arguments *{y 0.75 s 0.5}*. *y 0.75* va décaler le triangle de 0.75 vers le haut (axe des y). *S 0.5* multiplie la taille du triangle par 0.5, ce qui revient à diviser le triangle par 2. Cela veut dire que l'on va, à chaque fois que l'on appelle la règle *fractal*, dessiner un triangle deux fois plus petits que le triangle précédent en le décalant toujours de 0.75 sur l'axe des y.

Observons ensuite les deux instructions suivantes :

```
fractal {x 0.75 s 0.5 r -90}
fractal {x -0.75 s 0.5 r 90}
```

¹ Cette image et le code lui correspondant proviennent de cette adresse (de légères modifications ont été appliquées sur le code) : <http://www.contextfreeart.org/gallery/view.php?id=247>

Pour la première, la seul différence est qu'ici on décale le triangle de 0.75 sur l'axe des x et qu'on effectue une rotation de -90 degrés (ce qui est égale à une rotation de 270 degrés).

Pour la deuxième, on décale de -0.75 le triangle sur l'axe des x (soit du côté opposé à celui de la première instruction) et on effectue sur ce triangle une rotation de 90 degrés.

Les rotations fournies en arguments dans ces deux instructions permettent d'avoir des triangles dont un des sommets vise toujours un des deux axes x ou y.

Vous observez qu'à chaque fois qu'on dessine un nouveau triangle en le décalant vers le haut, on exécute les trois instructions qui sont présentes dans la règle *fractal*. C'est ainsi qu'on obtient toujours un triangle « relié » ou « connecté » à trois autres triangles.

Pour résumer simplement, voici comment se construit cette fractale :

1. On crée un simple triangle.
2. On dessine trois autres triangles, de la moitié de la taille du triangle précédent : le premier en dessus, le second à gauche et le troisième à droite du triangle de base. Pour les triangles dessinés à gauche et à droite, on effectue une rotation pour que le sommet du triangle soit dirigé vers un des axes (x ou y).
3. Pour chaque triangle ainsi obtenu, on fait la même chose : triangle en haut, à gauche et à droite...

Remarque : au moment de l'étape 3, lorsqu'on a obtenu un triangle à gauche ou à droite par exemple, et qu'on veut créer trois nouveaux triangles depuis celui-ci, c'est à dire un en haut, un à gauche et un à droite, l'ordinateur va créer les triangles en les décalant centralement par rapport au triangle précédent, et non pas par rapport aux axes x et y !

Afin de mieux comprendre cette remarque très importante, observez ce que va produire le code suivant, qui est légèrement différent du précédent.

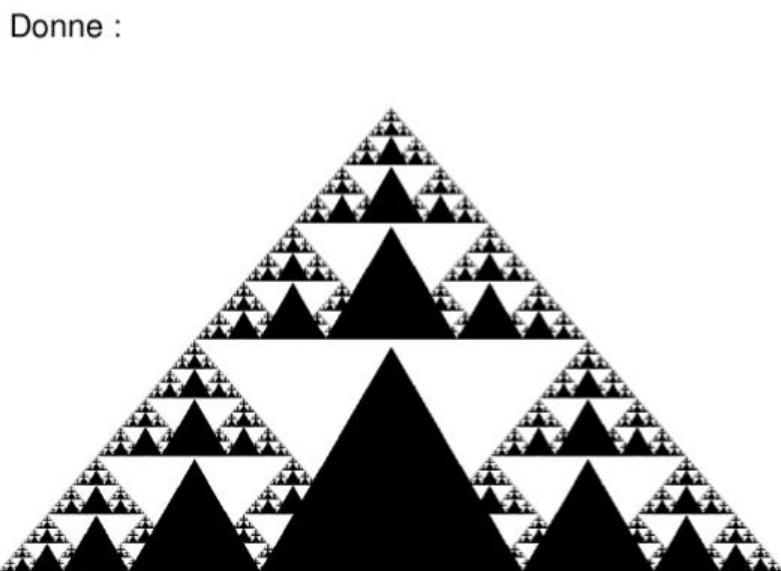
```
startshape fractal

rule fractal {
    TRIANGLE {}
    fractal {y 0.75 s 0.5}
    fractal { x 0.75 y -0.143 s 0.5}
    fractal { x -0.75 y -0.143 s 0.5}
}
```

On remarque qu'ici, on n'a pas effectué de rotation au triangle que l'on crée à gauche ($x 0.75$) et à droite ($x -0.75$) du triangle précédent. De plus, on a effectué un décalage en arrière sur l'axe des y de $-0,143$ pour les triangles de droites et de gauches.

C'est en fait pour bien les aligner sur la même droite.

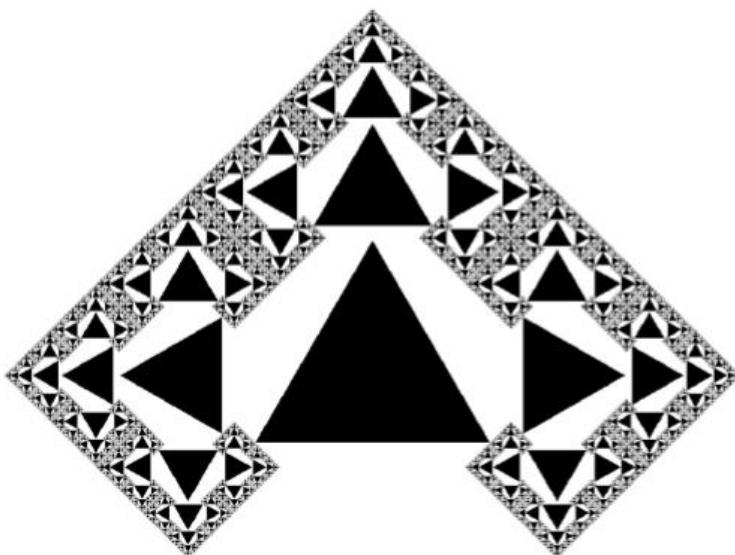
Si non, sans les deux arguments $y -0,143$ présents dans chacune des instructions permettant la création des triangles à gauche et à droite, nous obtiendrions :



Mais revenons à l'image obtenue précédemment, où les triangles sont bien alignés sur l'axe des y . Dans cette image, on a appliqué les mêmes appels de règles avec les mêmes arguments, excepté les arguments de rotation que l'on a omis. Cependant, on obtient une image assez différente.

En fait, c'est dans la toute première image sur les fractales que l'ordinateur réagit autrement lorsqu'il crée des triangles. Voici cette image (que nous avons vu il y a peu de temps) :

Jusqu'aux quatre triangles principaux (les quatre plus grands, dont un est plus grand que les trois autres), il n'y a pas de problème ; on a pris le triangle du milieu, divisé sa taille en deux pour créer des triangles à gauche, en haut et à droite, et n'oublions pas la rotation que l'on a effectuée sur les triangles de gauche et de droite pour que leur sommet pointent vers l'axe des y.



Cependant, à l'étape suivante, on peut se demander, lorsqu'on prend un des triangles de gauche ou de droite (peu importe lequel mais pas celui du haut), pourquoi est-ce que l'ordinateur créer des triangles, deux fois plus petits certes comme il le devrait, mais en les décalant cette fois-ci vers le bas, vers la droite et vers le haut ? Les arguments de nos instructions précisent pourtant de créer des triangles décalés vers la droite, la gauche et vers le haut...

Tout est question de rotation ici. En fait, quand l'ordinateur effectue une rotation à un triangle (ou à n'importe quelle figure), il effectue une même rotation aux axes x et y de la figure en question. Et chaque figure possède ses propres axes x et y, puisque sinon, dans notre image, le triangle du haut (au-dessus du triangle principale) ne donnerait pas ensuite le jour à des triangles se dirigeant eux aussi vers le haut.

Il est essentiel de bien comprendre ceci.

Pourquoi alors dans la seconde image où nous voyons les triangles en pyramides, les triangles correspondent tout le temps aux instructions avec les arguments que nous avons définis (c'est à dire que l'ordinateur crée toujours quelque que soit le triangle, un triangle à droite, un à gauche et un en haut) ? C'est parce qu'on a effectué de rotation à aucun des triangles, souvenez – vous ! Et qui dit pas de rotation de figure dit pas de rotation d'axe x et y...

3.3.2 Mélange de fractale et de système de règle

Et si maintenant nous tentions de mélanger ce que nous savons sur les fractales avec ce que nous avons appris lorsque nous avons poussé un peu plus loin le système de règle ?

```

startshape decaler_le_tout

rule decaler_le_tout{
    fractal{}
    decaler_le_tout {r 16 s 0.99 x 3}
}

rule fractal {
    TRIANGLE {}
    fractal { y 0.75 s 0.5}
    fractal { x 0.75 s 0.5 r -90}
    fractal { x -0.75 s 0.5 r 90}
}

```

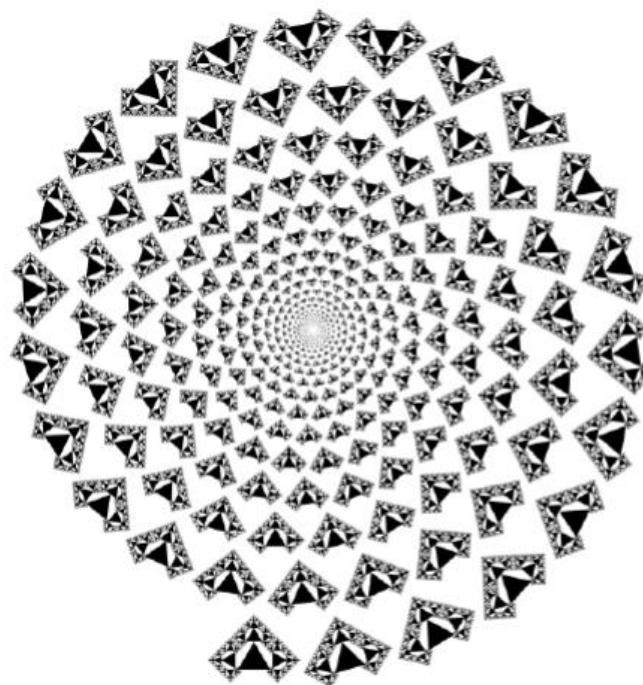
Donne :

L'analyse de cette figure ne devrait maintenant plus vous poser de problème.

La règle *fractal* est la même règle que la première règle sur les fractales que nous avons vu.

La règle *décaler_le_tout* ordonne à l'ordinateur de créer une fractale en appelant la règle *fractal* en la décalant de 3 sur l'axe des x, en effectuant sur cette figure une rotation de 16 degrés et en multipliant sa taille par 0.99 à chaque fois.

Admirez au passage l'effet visuel « palmé » des lignes blanches qui se trouvent entre chaque petite fractale...



3.3.3 Obtenir un tapis de Sierpiński avec des cercles

Voici un autre exemple de fractale.

Première étape :

```

startshape nice

rule nice{
    CIRCLE{}
    nice{x 2 s 0.5}
    nice{x -2 s 0.5}
}

```

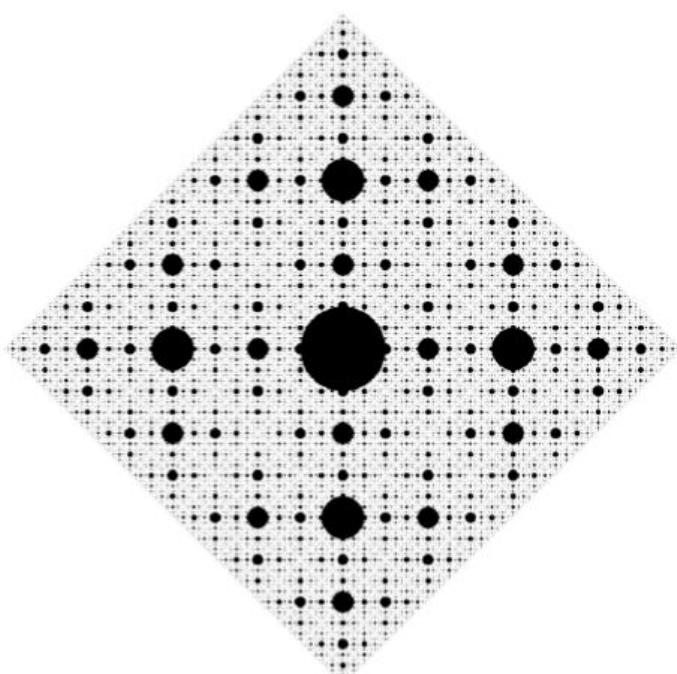
Donne :



En observant l'image, nous pourrions nous demander pourquoi il y a des cercles qui sont plus grands que d'autres mais qui se situent plus loin que certains petits cercles. En effet à chaque fois que l'on appelle la règle *nice*, on demande à l'ordinateur de créer 2 nouveaux cercles, un à gauche et un à droite, et on lui demande de diviser leur taille par deux à chaque fois. Théoriquement, on devrait donc créer des cercles de plus en plus petits. Mais sur notre image nous pouvons remarquer qu'on crée parfois des cercles plus grands que d'autres et pourtant ils sont plus éloignés que les cercles de plus petite taille.

C'est en fait tout à fait normal. Il ne faut pas oublier que l'ordinateur va toujours reprendre les derniers cercles construits et à partir de ces formes-là, recréer des nouveaux cercles. Étant donné qu'on crée des cercles que l'on décale à gauche ou à droite, certains grands cercles sont plus éloignés que d'autres cercles qui sont plus petits. Il ne faut pas oublier ceci afin de bien comprendre comment marche le programme.

Avec deux instructions supplémentaires, voici ce que nous pouvons facilement obtenir :



```
startshape nice
rule nice{
  CIRCLE{}
  nice{x 2 s 0.5}
  nice{x -2 s 0.5}
  nice{y 2 s 0.5}
  nice {y -2 s 0.5}
}
```

Donne, ci-contre :

Pas besoin d'explications, avec le code vous devriez facilement comprendre comment on arrive à obtenir une telle image. Noter qu'une ou deux lignes d'instructions suffisent parfois à créer dénormes différences avec Context Free Art.

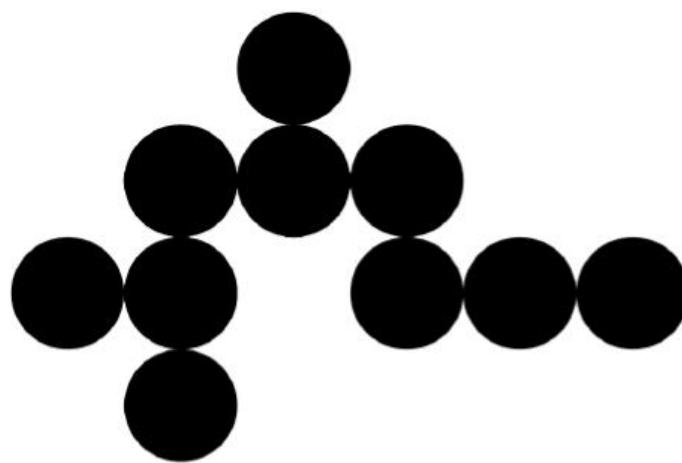
3.4 Utilisation du hasard

3.4.1 Élaboration d'un chemin de cercle

Il est possible d'utiliser le hasard avec Context Free Art. Grâce à cela, nous allons pouvoir créer des dessins qui seront par exemple décalé d'une certaine distance sélectionnée aléatoirement parmi plusieurs règles portant le même nom.
Pour mieux comprendre, voici un exemple de dessin utilisant le hasard.

```
startshape start  
  
rule start{  
    chemin{}  
}  
  
rule chemin{  
    CIRCLE{}  
    chemin{x 1}  
}  
  
rule chemin{  
    CIRCLE{}  
    chemin{x -1}  
}  
  
rule chemin {  
    CIRCLE{}  
    chemin{y 1}  
}  
  
rule chemin{  
    CIRCLE{}  
    chemin{y -1}  
}  
  
rule chemin{  
}
```

Donne, par exemple :



Étant donné que le hasard intervient, nous obtiendrons à chaque fois une image différente si l'on effectue plusieurs *render*.

Le code qui permet de dessiner ces cercles est relativement simple à comprendre. Il y a quatre règles nommées *chemin* : chacune permet de créer un cercle décalé à gauche, à droite, en haut ou en bas ($-x$, x , y , $-y$). Vous remarquerez qu'on a pris soin de donner le même nom à toutes ces règles : *chemin*. C'est comme ça que l'on peut utiliser du hasard avec context free art : lorsqu'il y a plusieurs règles partageant le même nom et qu'on appelle l'une d'entre elles, context free art va en choisir une au hasard (parmi celles qui portent le nom spécifié) et effectuer les instructions

présentes à l'intérieur. De cette façon, on arrive à créer un chemin tracé aléatoirement.

Maintenant, observez bien la dernière règle *chemin*. Vous pouvez constater qu'elle ne contient aucune instruction. Pourquoi alors avons-nous besoin de cette règle ? Tout simplement parce que sans elle, le programme continuerait de créer des cercles sans jamais s'arrêter, puisqu'à chaque fois on appelle la règle *chemin*. C'est pourquoi on crée une règle *chemin* sans aucune instruction : lorsque l'ordinateur choisit au hasard cette règle, il va s'arrêter de faire des cercles puisqu'il ne reçoit aucune instruction.

Dans notre cas, il y a donc 1 chance sur 5, à chaque fois que l'on dessine un cercle, que le programme s'arrête. Comment faire si nous voulons que l'ordinateur trace un plus long chemin, autrement dit pour qu'il y ait moins de chance que l'ordinateur choisisse d'appliquer la règle *chemin* ne contenant pas d'instructions ?

C'est très simple ; il suffit de placer un coefficient juste après le nom de la règle que l'on a écrite.

Regardez le code suivant qui montre bien comment placer le coefficient :

```
startshape start

rule start{
    chemin{}
}

rule chemin{
    CIRCLE{}
    chemin{x 1}
}

rule chemin{
    CIRCLE{}
    chemin{x -1}
}

rule chemin {
    CIRCLE{}
    chemin{y 1}
}

rule chemin{
    CIRCLE{}
    chemin{y -1}
}

rule chemin 0.01{}
```

Donne, par exemple :



On a placé le coefficient 0.01 juste après avoir écrit le nom de la règle à laquelle on voulait appliquer ce coefficient. Il y a maintenant 0.01 chance sur 5 pour que le programme s'arrête. Si on avait mis comme coefficient 0.5, on aurait 0.5 chance sur 5, ce qui revient à dire 2 chances sur 10, pour qu'il s'arrête.

On peut évidemment mettre un coefficient plus grand que 1 : en mettant un coefficient de 2 à la dernière règle, le programme s'arrêtera deux fois plus vite...

3.4.2 Rotation d'un angle aléatoire

Imaginons que l'on voudrait créer des flèches avec un angle aléatoire, compris entre 0 et 360. Nous pourrions très bien créer une règle pour chaque angle, mais cela prendrait beaucoup trop de temps puisqu'il faudrait alors écrire 360 règles.

Nous aurions donc quelque chose comme cela :²

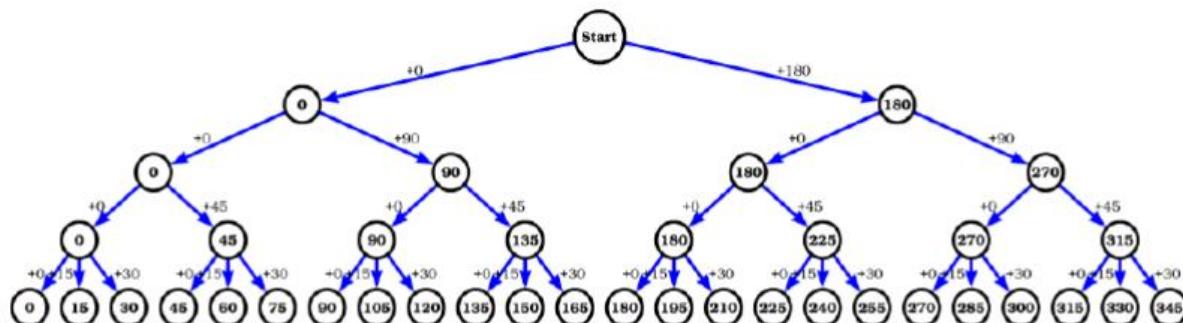
```
rule anglealea{
    fleche{r 1}
}
rule anglealea{
    fleche{r 2}
}
rule anglealea{
    fleche{r 3}
}
...
rule anglealea{
    fleche{r 359}
}
```

Il y a pourtant un moyen très simple qui va nous permettre d'obtenir des angles de 0 à 360. L'idée est d'abord de faire une rotation de 180 ou 0 degrés. Puis ensuite de tourner encore de 0 ou de 90 degrés et enfin de 0 ou de 45 degré...

Jusque-là, on déjà 8 angles possible (0, 45, 90, 135, 180, 225, 270, 315).

Ensuite, si on veut aller encore plus loin, on doit diviser l'angle que l'on ajoute par 3 à chaque fois, car si on divise par 2 on obtient des angles non-entiers.

Voici un schéma résumant bien la situation :



² L'idée, l'explication (que j'ai traduite), le code et l'image de cette page proviennent de cette source : http://www.contextfreeart.org/mediawiki/index.php/Tutorials/Rotating_by_a_random_angle

Cela nous donne 24 possibilités. Mais il est envisageable d'aller encore plus loin. Pour cela, il faut d'abord pour chaque angle trouvé effectuer une rotation de 0, 5 ou 10 degrés. Et finalement, en appliquant sur chaque angle obtenu une rotation de 0, 1, 2, 3 ou 4 degrés, on peut obtenir un angle allant de 0 à 360 degré !

Voyons maintenant comment mettre en place cette méthode avec context free art. Notre objectif étant de créer des flèches dirigées dans un angle aléatoire, nous allons commencer par créer une flèche.

Pour ceci, observez le code suivant :³

```
startshape fleche
rule fleche{
    SQUARE{x 0.5 size 1 0.075}
    TRIANGLE{x 1 size 0.2 r 30}
}
```

Donne :



On a simplement dessiné un carré puis un triangle. Vous remarquerez qu'on a utilisé, dans le cas du carré, 2 valeurs avec l'argument *size* : c'est une chose que nous n'avions pas encore faite jusqu'à maintenant mais c'est possible. En fait, l'argument *size* permet de changer la taille d'une forme, en *x* et en *y*. On peut évidemment multiplier la taille en *x* et en *y* avec un multiplicateur différent. Dans notre cas, on a écrit *s 1 0.075*, ce qui veut dire que la taille en *x* reste inchangée (1 *x*) alors qu'on multiplie la taille en *y* par 0.075, ce qui nous donne un rectangle. Lorsqu'on ne précise qu'une seule valeur avec l'argument *size*, c'est comme si on utilisait la même valeur pour multiplier la taille en *x* et celle en *y*.

On a ensuite utilisé *x 0.5* pour le carré et *x 1* pour le triangle ; il s'agit ici de coller la flèche au rectangle. On pourrait très bien obtenir la même forme en utilisant d'autres valeurs pour *x* mais il est alors possible que l'on ait des surprises au final : en effet il ne faut pas oublier que le programme zoomé sur l'image obtenue, ce qui donne l'impression d'avoir la même flèche par exemple en utilisant *x 0* pour le carré et *x 0.5* pour le triangle, cependant on obtient une flèche deux fois plus grande ; seulement on ne peut pas le remarquer puisque le programme effectue un zoom.

Essayons maintenant de créer une flèche avec un angle pris au hasard. C'est ici que nous allons utiliser notre méthode spéciale pour les angles.

³ Ce code provient de la source suivante :

http://www.contextfreeart.org/mediawiki/index.php/Tutorials/Rotating_by_a_random_angle

Observez le code suivant :⁴

```
startshape RandomRotate1

rule DrawArrow {
    SQUARE {x 0.5 size 1 0.075 }
    TRIANGLE {x 1 r 30 size 0.2 }
}

rule RandomRotate1 {
    RandomRotate2 { }
}
rule RandomRotate1 { RandomRotate2 { r 180 } }
rule RandomRotate2 { RandomRotate3 { } }
rule RandomRotate2 { RandomRotate3 { r 90 } }
rule RandomRotate3 { RandomRotate4 { } }
rule RandomRotate3 { RandomRotate4 { r 45 } }
rule RandomRotate4 { RandomRotate5 { } }
rule RandomRotate4 { RandomRotate5 { r 15 } }
rule RandomRotate4 { RandomRotate5 { r 30 } }
rule RandomRotate5 { RandomRotate6 { } }
rule RandomRotate5 { RandomRotate6 { r 5 } }
rule RandomRotate5 { RandomRotate6 { r 10 } }
rule RandomRotate6 { RandomRotate7 { } }
rule RandomRotate6 { RandomRotate7 { r 1 } }
rule RandomRotate6 { RandomRotate7 { r 2 } }
rule RandomRotate6 { RandomRotate7 { r 3 } }
rule RandomRotate6 { RandomRotate7 { r 4 } }
rule RandomRotate7 { DrawArrow { } }
```

Tout d'abord, remarquez que vous pouvez très bien écrire une règle sur une seule ligne. Mais cela rend alors votre code beaucoup moins lisible que si vous faites des retours à la ligne comme je le fais depuis le début. Les conséquences d'un code peu lisible sont, d'une part qu'une autre personne que vous aura beaucoup de mal à comprendre votre code et, d'autre part, que vous aurez peut-être du mal à vous relire si vous essayez de modifier votre code ultérieurement.

Si j'ai ici laissé ces règles sous cette forme-là, c'est pour vous montrer que c'est une solution certes déconseillée mais possible et aussi parce que les valeurs des angles sont ainsi plus visibles. J'ai mis la première de ces règles sous la forme préférable, à titre d'exemple.

⁴ Le code est tiré du lien suivant (Je n'y ai modifié qu'une seule règle) :
http://www.contextfreeart.org/mediawiki/index.php/Tutorials/Rotating_by_a_random_angle

On commence donc avec la règle *RandomRotate1*. Il y a deux règles *RandomRotate1*: l'une effectue une rotation de 180 degrés et appelle la règle *RandomRotate2*. L'autre ne fait qu'appeler la règle *RandomRotate2*. Il existe deux règles *RandomRotate2*. L'une effectue une rotation de 90 degrés (ce qui ajoute 90 degrés à la rotation total à effectuer) et l'autre n'effectue pas de rotation. Chacune de ces deux règles appelle la règle *RandomRotate3*... Et ainsi de suite, jusqu'à ce qu'on appelle la règle *RandomRotate7* qui est unique et qui appelle la règle *DrawArrow*. Noter que la règle *RandomRotate7* contient le total des degrés ajoutés depuis le début, étant donné que lorsqu'on effectue une rotation on appelle la règle *RandomRotate* suivante avec comme argument le nombre de degré à ajouter. Et voilà ! En faisant un rendu, nous obtenons une flèche, comme précédemment, mais cette fois-ci avec un angle aléatoire.

Maintenant, essayons de dessiner plusieurs flèches comme celle-là. Pour ce faire, il suffit d'ajouter une règle :

```
rule DrawRandomArrows {
    30*{} RandomRotate1 { }
}
```

Comme nous l'avons vu, *RandomRotate1* permet de créer une flèche avec un angle aléatoire. Ici, on crée une règle qui appelle 30 fois la règle *RandomRotate1*. C'est à cela que sert le *30*{}* placer devant. On va donc créer 30 flèches. Si on ne voulait en créer que 10, il faudrait mettre *10*{}* au lieu de *30*{}*. Noter qu'on peut fournir un argument, c'est pour ça qu'il y a des accolades. Même sans argument, ces accolades sont essentielles pour le bon fonctionnement du code. On aurait pu évidemment écrire 30 fois *RandomRotate1{}* mais cela aurait pris beaucoup trop de temps.

Voici donc le code final⁵:

```
startshape DrawRandomArrows

rule DrawArrow {
    SQUARE {x 0.5 size 1 0.075 }
    TRIANGLE {x 1 r 30 size 0.2 }
}

rule RandomRotate1 {
    RandomRotate2 { }
}
rule RandomRotate1 { RandomRotate2 { r 180 } }
rule RandomRotate2 { RandomRotate3 { } }
rule RandomRotate2 { RandomRotate3 { r 90 } }
```

⁵ Code provenant de la page suivante :

http://www.contextfreeart.org/mediawiki/index.php/Tutorials/Rotating_by_a_random_angle

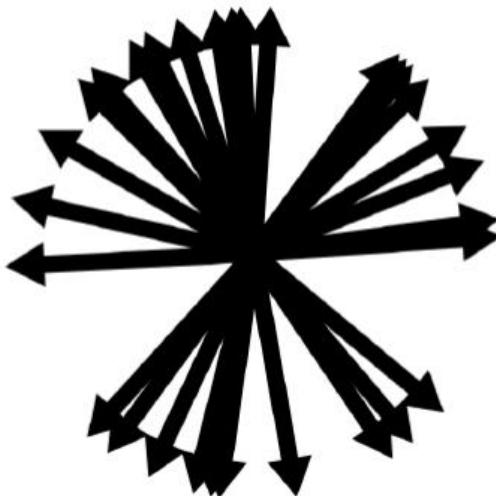
```

rule RandomRotate3 { RandomRotate4 { } }
rule RandomRotate3 { RandomRotate4 { r 45 } }
rule RandomRotate4 { RandomRotate5 { } }
rule RandomRotate4 { RandomRotate5 { r 15 } }
rule RandomRotate4 { RandomRotate5 { r 30 } }
rule RandomRotate5 { RandomRotate6 { } }
rule RandomRotate5 { RandomRotate6 { r 5 } }
rule RandomRotate5 { RandomRotate6 { r 10 } }
rule RandomRotate6 { RandomRotate7 { } }
rule RandomRotate6 { RandomRotate7 { r 1 } }
rule RandomRotate6 { RandomRotate7 { r 2 } }
rule RandomRotate6 { RandomRotate7 { r 3 } }
rule RandomRotate6 { RandomRotate7 { r 4 } }
rule RandomRotate7 { DrawArrow { } }

rule DrawRandomArrows {
  30*{} RandomRotate1 { }
}

```

Qui donne, par exemple :



Remplacer maintenant la règle *DrawRandomArrows* par celle-ci :

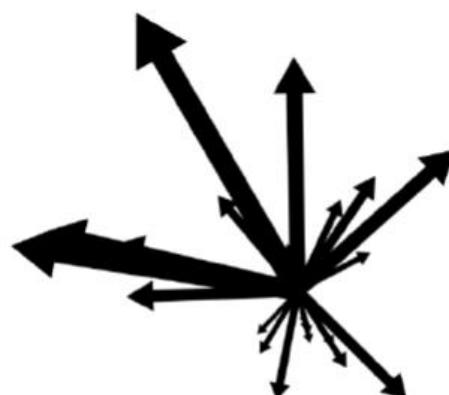
```

rule DrawRandomArrows {
  30*{s 0.9} RandomRotate1 { }
}

```

Les flèches dessinées sont de plus en plus petites. Normal, puisqu'on a fourni un argument lorsqu'on a créé 30 flèches. On a ajouté l'argument *s 0.9* entre accolades. À chaque fois que l'on crée une flèche, on multiplie sa taille par 0.9.

Donne, par exemple :



3.5 Utilisation des chemins

L'utilisation de *chemin* (*Path*, en anglais) va nous permettre de créer de nouvelles formes que nous allons ensuite pouvoir utiliser dans nos dessins.

Le carré, le triangle et le cercle sont en fait des formes construites avec la méthode *path*. Ce sont des formes déjà construites et vous n'avez pas à les recréer.

Lorsqu'on utilise la méthode *path*, c'est un peu la même chose qu'avec la méthode des règles : on crée une règle à laquelle on donne un nom. Sauf qu'ici, c'est un chemin et non une règle ; c'est pourquoi vous n'écrivez pas *rule* suivi du nom de la règle mais *path*, suivi du nom que vous voulez donner à ce chemin.

Les chemins contiennent eux-aussi des instructions que l'ordinateur va lire et exécuter. Cependant vous ne pouvez pas utiliser les mêmes types d'instructions dans les règles et dans les chemins. Dans les chemins, vous ne pouvez pas par exemple utiliser l'instruction *SQUARE{}* , car elle ne s'applique que dans les règles, et vice versa. Les chemins utilisent des instructions spéciales, différentes de celle présentes dans les règles, mais néanmoins le fonctionnement principal reste le même.

Autre remarque importante : il n'est pas possible de créer deux ou plusieurs chemins partageant le même nom.

Une dernière chose : lorsque vous trouvez dans une règle, vous pouvez appeler un chemin, mais le contraire n'est pas possible.

3.5.1 création de lignes et de polygones

Pour le moment, veuillez tester le code ci-dessous :

```
startshape ligne  
  
path ligne{  
    MOVETO{x 0 y 0}  
    LINETO{x 1 y 0}  
    STROKE{}  
}
```

Donne :



On commence donc avec le chemin *ligne* (On peut dire que les chemins sont des types de règle, pour ne pas confondre avec autre chose).

La première instruction *MOVETO {x 0 y 0}* ordonne à l'ordinateur de se déplacer jusqu'en (0 ; 0), sans dessiner (comme si on levait le crayon pour se déplacer sur une feuille). On aurait pu préciser un autre point.

La deuxième instruction, *LINETO {x 1 y 0}*, demande cette fois à l'ordinateur de tracer une ligne, en se déplaçant jusqu'en (1 ; 0).

La dernière instruction prend son importance surtout lorsqu'on dessine des polygones. Comme ici on ne trace qu'une simple ligne, on ne va pas tenter de comprendre ce que fait l'instruction *STROKE{}*. Nous verrons ça plus tard. Pour le moment, dites-vous juste que sans cette instruction, l'ordinateur ne trace pas de ligne.

À pars cette dernière instruction, il n'y a pas de problème jusqu'ici.

Essayons maintenant de créer un carré à l'aide d'un chemin.

C'est très simple. Observez le code suivant :

```
startshape carre

path carre{
    MOVETO{x 0 y 0}
    LINETO{x 1 y 0}
    LINETO{x 1 y 1}
    LINETO{x 0 y 1}
    CLOSEPOLY{}
    STROKE{}
}
```

Donne :



Si vous regardez bien, les deux premières instructions présentes dans le chemin *carre* sont les mêmes que celles utilisées pour construire notre ligne.

En fait, on commence en (0 ; 0). Puis on trace une ligne jusqu'en (1 ; 0). Ensuite, on en trace une autre jusqu'en (1 ; 1). Et on trace encore une ligne, jusqu'en (0 ; 1).

Pour la dernière ligne, on utilise l'instruction *CLOSEPOLY{}*. Lorsqu'on construit un polygone et qu'on utilise cette instruction, l'ordinateur trace une ligne depuis le point où il s'est arrêté la dernière fois jusqu'au point préciser dans l'instruction *MOVETO{}*.

Dans notre cas, on aurait évidemment pu tracer la dernière ligne nous-même, en ajoutant l'instruction *LINETO{x 0 y -0.05}* (-0.05 car sinon la ligne dépasse l'autre).

Essayez vous-même dans context free art si vous ne comprenez pas et vous verrez très vite ce qui ne fonctionne pas. Essayez de voir depuis quel endroit l'ordinateur commence à tracer les lignes et vous devriez comprendre). C'est cependant plus pratique et plus sûr d'utiliser l'instruction *CLOSEPOLY{}*, car l'ordinateur sait qu'il vient de terminer de dessiner un polygone avec cette instruction.

Essayez maintenant de remplacer l'instruction *STROKE{}* par *FILL{}*.

```
startshape carre
path carre{
    MOVETO{x 0 y 0}
    LINETO{x 1 y 0}
    LINETO{x 1 y 1}
    LINETO{x 0 y 1}
    CLOSEPOLY{}
    FILL{}
}
```

Donne :



Vous comprenez maintenant la différence entre *STROKE{}* et *FILL{}* : *FILL{}* remplit le carré (on peut ajouter un argument pour changer la couleur et la luminosité) tandis que *STROKE{}* ne dessine que les contours de la forme concernée.

Voyons au passage un nouvel argument dont on pourra se servir dans nos prochains dessins : l'argument *brightness*. On peut simplement écrire *b* lorsqu'on l'utilise, pour aller plus vite. En français, *brightness* veut dire « luminosité ».

Essayer par exemple le code suivant :

Donne :

```
startshape carre
background{b -1}

path carre{
    MOVETO{x 0 y 0}
    LINETO{x 1 y 0}
    LINETO{x 1 y 1}
    LINETO{x 0 y 1}
    CLOSEPOLY{}
    FILL{b 1}
}
```

On obtient un carré blanc sur un fond noir (On ne voit que le bord du fond noir sur l'image) :

Regarder bien ce qu'il est écrit après la première ligne : *background {b -1}*.

Background est en fait une directive qui permet de modifier la couleur du fond sur lequel le logiciel dessine. Accompagnée de l'argument *b -1*, cette directive ordonne à l'ordinateur de créer un fond noir.

Quant à l'instruction `FILL{}`, on lui a ajouté l'argument `b` qui règle la luminosité sur la valeur 1. Cela a pour effet de remplir le carré avec du blanc.

On sait donc que lorsque l'argument `b` vaut -1, on aura du noir, et lorsqu'il vaut 1, on aura du blanc.

On peut utiliser des valeurs pour `b` qui se trouve entre 1 et -1. On obtiendra alors diverses sortes de gris.

Expérimitez le phénomène par vous-même en changeant les valeurs de `b` dans `FILL{}` et en mettant à la place 0.7 ou 0.4 par exemple.

3.5.2 Expérience et contrôle de l'épaisseur du trait

Essayons maintenant une petite expérience. Nous allons d'abord créer une règle qui va dessiner des carrés de plus en plus petits.

Voici ce code :

```
startshape dessin
rule dessin{
  SQUARE{}
  dessin{x 1.1 s 0.5}
}
```

Donne :



Rien de surprenant.

Essayons maintenant de faire la même chose, mais cette fois-ci en créant le carré nous-mêmes, à l'aide d'un chemin.

Voici le code que cela donnerait :

```
startshape dessin
path carre{
  MOVETO{x 0 y 0}
  LINETO{x 1 y 0}
  LINETO{x 1 y 1}
  LINETO{x 0 y 1}
  CLOSEPOLY{}
}
rule dessin{
  carre{}
  dessin{x 1.1 s 0.5}
}
```

Résultat obtenu :



Surprenant ! Ici l'ordinateur ne centre pas les carrés comme il le fait dans l'image précédente. Pourtant le chemin `carre` crée un carré normal, qui est exactement le même que celui qu'on obtient avec l'instruction `SQUARE{}` placée dans une règle.

En fait, la différence n'est pas là.

Quand on a créé notre chemin, on a placé en premier l'instruction *MOVETO{x 0 y 0}*. Cela veut dire qu'à chaque fois qu'on crée un carré, on revient au point (0 ; 0). C'est pour cela que dans la deuxième image obtenue, les carrés sont tous alignés sur l'axe des x ! Et n'oubliez pas que comme on décale chaque carré de 1.1, leur axe x est aussi décalé de 1.1 aussi, sinon tous les carrés seraient entassés les uns sur les autres.

Le fait d'enlever l'instruction *MOVETO{x 0 y 0}* n'y change rien, vous pouvez tester par vous-même.

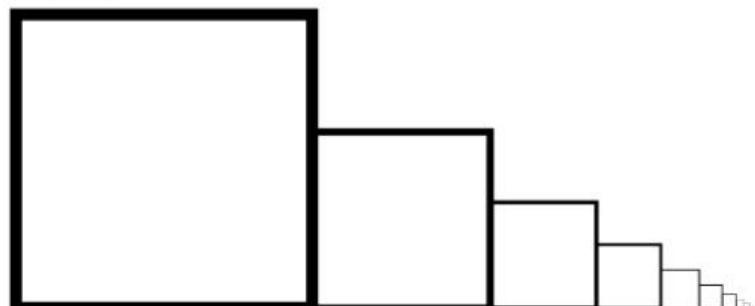
Enfin, dernière chose : on peut utiliser un autre argument avec l'instruction *STROKE{}*. Il s'agit de l'argument *width*, qui ne possède pas de raccourci. Il s'agit ici de l'épaisseur du trait que l'ordinateur va tracer. Cet argument est aussi disponible avec l'instruction *FILL{}*, même si cela ne sert à rien (puisque *FILL{}* remplit un polygone).

Par exemple :

```
startshape faire_carre
rule faire_carre{
    carre{}
    faire_carre{x 1 s 0.6}
}

path carre{
    MOVETO{x 0 y 0}
    LINETO{x 1 y 0}
    LINETO{x 1 y 1}
    LINETO{x 0 y 1}
    CLOSEPOLY{}
    STROKE{width 0.04}
}
```

Donne :



Remarquer que les carrés ne sont pas tout à fait alignés sur l'axe des x. En fait, c'est parce que l'ordinateur aligne le milieu du trait de chaque carré sur la même ligne. Comme l'épaisseur du trait change (puisque la taille du carré est à chaque fois multipliée par 0.5), le milieu du trait se modifie aussi et l'alignement effectué par l'ordinateur fait que les carrés ne seront pas tous sur la même ligne.

En fait, ils le sont, mais par rapport au milieu de leur trait.

3.5.3 Arc de cercles

Il existe d'autres types d'instructions pour effectuer des déplacements. Nous allons par exemple voir comment construire des arcs de cercles avec Context Free Art.

Cependant nous verrons juste comment créer un arc de cercle simple, car la construction d'arc de cercle se révèle assez compliquée dans Context Free Art.

Pour faire un arc de cercle, on doit d'abord créer un chemin. Ensuite, on utilise l'instruction ARCTO{}, avec au moins deux arguments : un argument x ou y suivit d'un nombre, pour la distance jusqu'à laquelle on va tracer l'arc de cercle, et un argument r pour l'angle.

Comme je l'ai dit plus tôt, nous n'irons pas plus loin en ce qui concerne la construction d'un demi-cercle.

Observez le code suivant :

```
startshape demicercle
path demicercle{
    MOVETO {x 0 y 0}
    ARCTO{x 3 r 1}
    STROKE{}
}
```

Donne :



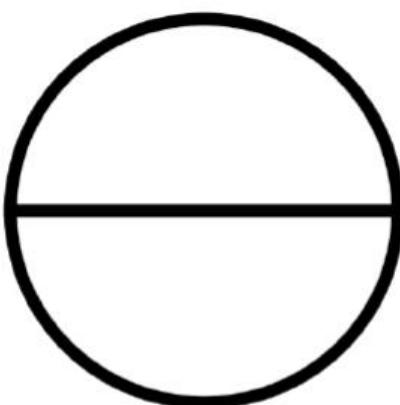
En bref, on crée un chemin dans lequel on commence par dire à l'ordinateur d'aller en (0 ; 0). On lui demande ensuite de tracer un arc de cercle, avec les arguments {x 3} et {r 1}.

Nous allons maintenant nous intéresser à un nouvel argument : l'argument *flip*. Lorsqu'on l'utilise, on peut soit écrire *flip* ou soit écrire *f*, qui est un raccourci. Puis on écrit ensuite le nombre, qui sera l'angle utilisé. En fait, *flip* fait un effet miroir : il sert à dessiner une forme à l'envers. Et l'angle utilisé avec l'argument *flip* sera celui de ce « miroir ».

Voici tout de suite un exemple pour bien comprendre.

Le premier en utilisant l'argument {f 0}, qui provoque un effet de miroir à l'horizontale.

Résultat obtenu :



```
startshape fairedemi
rule fairedemi{
    demicercle{}
    demicercle{f 0}
}

path demicercle{
    MOVETO {x 0 y 0}
    ARCTO{x 3 r 1}
    CLOSEPOLY{}
    STROKE{}
}
```

Noter qu'on a utilisé l'instruction `CLOSEPOLY{}` afin de créer des demi-cercles.

Maintenant, observer le code suivant, où on utilise l'argument `flip` avec un angle de 90 degrés :

```
startshape fairedemi

rule fairedemi{
    demicercle{}
    demicercle{f 90}
}

path demicercle{
    MOVETO {x 0 y 0}
    ARCTO{x 3 r 1}
    CLOSEPOLY{}
    STROKE{}
}
```

Donne :



Cela nous donne un effet miroir à la verticale !

Dans les deux dessins que nous venons de voir, on peut dire qu'on a effectué une symétrie axiale, une fois d'axe x (horizontale), avec comme valeur d'angle 0, et l'autre fois avec une valeur d'angle de 90 degrés, ce qui provoque une symétrie axiale d'axe y.

L'argument `flip` est parfois très pratique, il évite d'avoir à réécrire certaines règles et permet grâce à son effet miroir de nous faire gagner du temps.

3.5.4 Synthèse des derniers points

Voici un dessin qui reprend un peu tout ce qu'on a vu dernièrement, c'est-à-dire l'utilisation des arguments `brightness`, `flip`, et la construction de chemin en utilisant des arcs de cercles. Essayez de comprendre comment fonctionne le code suivant.

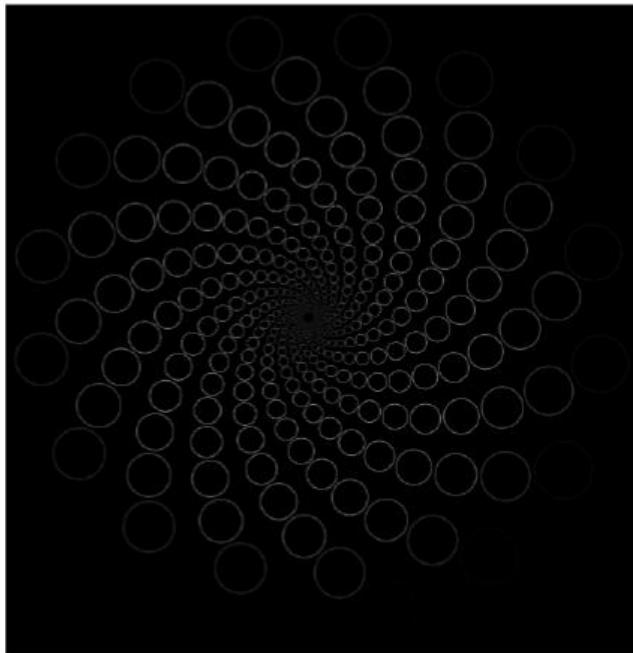
```
startshape fractal

background{b -1}

rule fractal{
    demicercle{}
    demicercle{f 0}
    fractal{x 6 s 0.99 r 22 b 0.01}
}

path demicercle{
    MOVETO {x 0 y 0}
    ARCTO{x 3 r 1}
    STROKE{}
}
```

Donne :



Analysons le code.

Il faut tout d'abord remarquer qu'on a placer une *directive*, il s'agit de *background{b -1}*, ce qui crée un fond noir.

On remarque aussi plus loin, dans le chemin *demi-cercle*, qu'on va utiliser l'instruction *STROKE{}* sans argument. Le programme considère dans ce cas que la valeur de l'argument *brightness* vaut 0.

Puis on remarque aussi que dans la règle *fractal*, la dernière instruction appelle la règle *fractal* avec divers argument dont *b 0.01*. Les cercles que l'on crée d'abord en noir deviendront de plus en plus gris puis blanc à mesure que la valeur de *b* s'approche de 1.

Comment fait-on ensuite pour créer ces cercles aux contours blancs mais aux fonds noirs ? Car si on avait utilisé l'instruction *CIRCLE{}*, on n'aurait pas pu utiliser les instructions *FILL{}* et *STROKE{}*, qui ne peuvent être utilisées que dans les chemins. Avec l'instruction *CIRCLE{}*, réaliser le même dessin que dans cet exemple est possible, mais il faut pour cela construire un cercle blanc puis en dessiner un autre plus petit, de fond noir, par-dessus.

Dans notre exemple, on a procédé autrement :

On crée les cercles dans la règle *demicercle*. Pour cela on utilise l'instruction *ARCTO{}*. On choisit ensuite de ne remplir que le contour du cercle, grâce à la commande *STROKE{}*. Cela crée un demi-cercle.

L'autre partie du cercle est créée lorsqu'on appelle la règle *fractal*, qui est la règle par laquelle commence le programme. Dans cette règle, on appelle une première fois la règle *demicercle* sans argument, puis on l'appelle une seconde fois avec l'argument *f 0*. Rappelez-vous que *f* est le raccourci de *flip*. Ce qui nous permet de dessiner un arc de cercle, mais à l'envers. On obtient donc un cercle !

Puis on appelle la règle *fractal*, avec divers arguments.

Il y a beaucoup d'autres formes que vous pourrez créer grâce aux arcs de cercles. Laissez place à votre imagination.

3.6 Utilisation de couleurs

Maintenant que nous avons vu comment faire pas mal de choses avec Context Free Art, nous allons nous intéresser aux couleurs.

Il est bien évidemment possible d'utiliser des couleurs avec Context Free Art.

Pour ce faire, nous avons à disposition 3 arguments :

- L'argument *hue*, abrégé *h*
- L'argument *saturation*, abrégé *sat* (puisque le raccourci *s* existe déjà, pour *size*)
- L'argument *brightness*, abrégé *b* (que nous connaissons déjà !)

L'argument *brightness*, permet de régler la luminosité. Plus il est proche de 1 et plus la luminosité est élevée. Plus il est proche de 0 est plus on obtiendra une couleur sombre.

Les arguments *hue* et *saturation*, quant à eux, permettent d'obtenir une couleur.

La valeur de l'argument *hue* peut varier de 0 à 360.

La valeur de l'argument *saturation* peut varier de 0 à 1 (Jusqu'à quatre chiffres après la virgule. Exemple : 0,4456, 0.9613...)

Il est difficile d'expliquer comment la couleur varie en fonction de ces trois arguments. Retenez simplement que *brightness* contrôle la luminosité et que les deux autres arguments servent à définir la teinte de la couleur.

Rappelez-vous qu'un outil, compris dans le logiciel, permet de rapidement trouver quelles valeurs il faut assigner à ces paramètres pour obtenir une couleur particulière. Il suffit d'aller sous l'onglet « Window » du programme, puis de cliquer sur « Color calculator ». Pour savoir comment utiliser cet outil, se référer à la section correspondante.

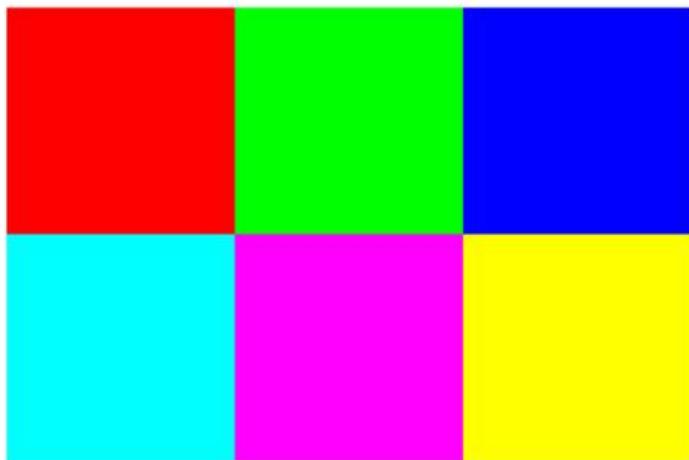
3.6.1 Quelques couleurs de bases :

Voici comment obtenir quelques couleurs de bases. Nous les avons utilisées sur un carré, mais un cercle ou un triangle aurait très bien pu faire l'affaire.

```
startshape carrecouleur

rule carrecouleur{
    SQUARE{sat 1 hue 0 b 1} //Rouge
    SQUARE{sat 1 hue 120 b 1 x 1} //Vert
    SQUARE{sat 1 hue 240 b 1 x 2} //Bleu
    SQUARE{sat 1 hue 180 b 1 y -1} //Cyan
    SQUARE{sat 1 hue 300 b 1 x 1 y -1} //Magenta
    SQUARE{sat 1 hue 60 b 1x 2 y -1} //Jaune
}
```

Donne (page suivante) :



Remarquez que pour ces couleurs de bases, seul l'argument *hue* change. L'argument *saturation* ne change que pour des couleurs plus spécifiques.

Notez également l'utilisation de commentaires. En programmation, il est important d'utiliser du commentaire pour pouvoir par la suite plus facilement continuer son

projet ou pour que quelqu'un puisse comprendre comment fonctionne un programme lorsque ce n'est pas lui qu'il l'a écrit.

Dans Context Free Art, il suffit d'écrire // avant de rédiger un commentaire. Tout ce qui se trouve après ces deux barres obliques ne sera pas lu par l'ordinateur. C'est exactement ce que j'ai fait dans le code que nous venons de voir.

3.6.2 Utilisation d'un dégradé :

Essayons maintenant de varier la teinte d'une forme pour créer un dégradé.

Pour créer la forme, on va dessiner un carré, puis le redessiner toujours de plus en plus petit en appliquant sur lui un angle et en le décalant d'un côté. On aura donc une spirale. C'est très simple et nous l'avons déjà fait plusieurs fois.

Maintenant, pour ce qui est du dégradé, il s'agit de faire varier les arguments qui définissent la couleur durant la répétition du carré. C'est-à-dire que lorsqu'on va appeler la règle qui crée un carré simple, on ne va pas seulement lui fournir un argument *size*, un argument *r* et un argument *x*, mais on va en plus lui fournir un argument qui sert à définir une couleur, comme *hue* ou *saturation*. On peut également lui fournir l'argument *brightness*, dans ce cas c'est la luminosité qui changera.

Observez donc le code suivant :

```
startshape repetition

rule carrecouleur{
    SQUARE{ }
}

rule repetition{
    carrecouleur{sat 0.01 hue 0 b 1}
    repetition{x 0.2 sat 0.01 s 0.99 r 3}
}
```

Donne :



Si vous observez bien le code, on crée d'abord un simple carré.

Puis, dans la règle *repetition*, on appelle la règle qui crée ce carré avec divers argument, dont l'argument *sat* avec la valeur 0.01. On attribue une valeur de 0 pour l'argument *hue* et une valeur de 1 pour l'argument *brightness*. Pour rappel, lorsqu'on veut obtenir du rouge, il faut simplement une valeur de 1 pour *brightness* et *saturation* et une valeur de 0 pour *hue*.

Cela crée un carré légèrement rosé. Puis la règle *repetition* s'appelle elle-même, avec divers arguments dont l'argument *sat* suivit de la valeur 0.01. Cela permet d'augmenter la valeur de la saturation au fur et à mesure que l'on crée des carrés. Et voilà notre dégradé !

3.6.3 Une autre utilisation de la couleur

Voyons maintenant une autre application de la variation de la couleur.

Observez ce que produit le code suivant :

```
startshape loop

rule loop{
  carp{}
  loop{x 0.3 y 0.3 r 7 b 0.6 sat 14 hue 30}
}

rule loop{
  carp{}
  loop{x 0.3 y 0.3 r -7}
}

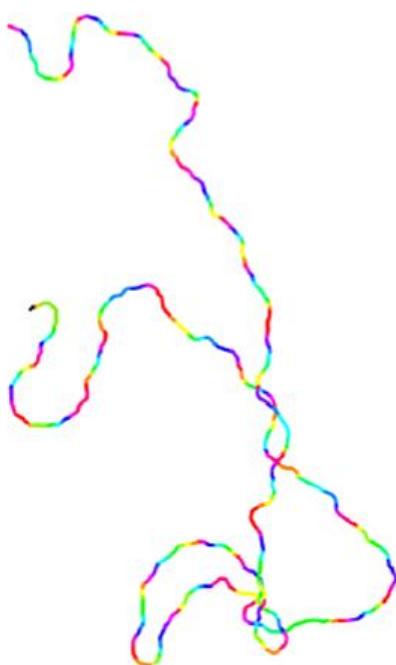
rule loop{
  carp{}
  loop{x 0.3 y 0.3 r -14}
}

rule loop{
  carp{}
  loop{x 0.3 y 0.3 r 14}
}

rule loop 0.008{

rule carp{
  CIRCLE{}
}
```

Donne, par exemple :



Analysons ce code. Il faut d'abord regarder la toute dernière règle, la règle *carp*. Elle ne sert qu'à créer un simple cercle.

Puis on a plusieurs règles *loop*. Et dans chacune de ces règles, on fait la même chose : on appelle la règle *carp*, puis on appelle la règle *loop* avec divers arguments. Seul l'argument *r* varie d'une règle à une autre. Il y a quatre règles *loop* (sans compter celle qui ne contient aucune instruction), et dans chacune l'argument *r* vaut soit 7, -7, 14 ou -14.

Comme on a le même nom pour plusieurs règles, l'ordinateur choisit l'une d'elles au hasard (sauf pour la cinquième règle *loop*, qui a moins de chance d'apparaître puisque on a mis la valeur 0.008. Lorsque l'ordinateur tombe sur cette règle, il s'arrête de dessiner). C'est pour cela que l'ordinateur va tracer un chemin qui peut très bien aller à droite, à gauche, en haut ou en bas. Si vous faites un rendu, vous obtiendrez à chaque fois une image différente.

Pour ce qui est de la couleur, on a simplement inséré dans une des règles *loop* quelques arguments spécifiques aux couleurs avec des valeurs prises arbitrairement (Il s'agit de la règle *loop* qui contient l'instruction *loop{x 0.3 y 0.3 r 7 b 0.6 sat 14 hue 30}*). Ainsi, lorsque l'ordinateur tombe sur la règle *loop* qui contient ces arguments, la couleur des formes dessinées va changer.

3.7 Projet finale : construction d'un arbre

Maintenant que nous avons vu comment réaliser pas mal de choses avec Context Free Art, nous allons essayer de construire un arbre. Nous devrions normalement être capables, puisque nous n'aurons besoin que de ce que nous avons déjà vu ; de la récursivité pour les branches, du hasard afin d'obtenir un arbre avec des branches qui se dirigent aussi bien à droite qu'à gauche, de la couleur pour les feuilles et le tronc, etc...

Il est donc évident qu'une fois que nous aurons terminé, nous obtiendrons un arbre différent après chaque *render*, puisque nous allons utiliser le hasard.

En fait nous irons même jusqu'à élaborer une forêt d'arbres. C'est simple, une fois que nous aurons construit un premier arbre, nous n'aurons qu'à créer une règle qui va construire un arbre et ensuite faire appeler cette règle elle-même avec divers arguments.

Mais pour le moment, essayons déjà de construire un arbre. Nous allons y aller progressivement ; tout d'abord nous allons créer un simple tronc, ensuite une feuille, puis un arbre, suivi de la couleur et pour finir nous construirons une forêt.

3.7.1 Étape 1 : Le tronc

Pour commencer, il nous faut un tronc, qui penche légèrement d'un côté de préférence. Rien de plus simple : on va créer des cercles, les répéter avec divers arguments, comme quand on cherche à créer une spirale, avec un angle moins élevé tout de même.

Donne :

```
startshape tree

rule tree {
    CIRCLE {s 0.4 1}
    tree { s 0.96 y 0.33 r 1.4}
}
```



Voilà. Nous n'analyserons que rapidement le fonctionnement de ce code. Observer-le et vous devriez comprendre. Juste un rappel : quand on crée un cercle avec l'argument *size* muni des valeurs 0.4 et 1, cela veut dire qu'on veut créer un cercle qui soit un peu moins large que haut (donc cela donne un ovale). Rappelez-vous que l'argument *size* peut très bien affecter les valeurs en *x* que celle en *y*.

En revanche, prêtons attention aux valeurs quelques instants. Pourquoi ces valeurs ? Au début on ne sait pas vraiment quelle valeur on va mettre, c'est une fois qu'on a obtenu un arbre qu'on change un peu ces valeurs pour trouver celles qui vont le mieux. C'est un peu du tâtonnement. J'y reviendrai à la fin. Si j'ai directement mis les valeurs finales c'est pour éviter de vous troubler en cours de route ; pour le moment, ne retenez que l'essentiel : on a un tronc qui penche légèrement de côté.

3.7.2 Étape 2 : le tronc qui penche des deux côtés

Comme je l'ai dit, nous allons y aller progressivement. Cette étape n'apporte qu'un simple changement : le tronc penche des deux côtés.

```
startshape tree

rule tree {
    CIRCLE {s 0.4 1}
    tree { s 0.96 y 0.33 r 1.4}
}

rule tree {
    CIRCLE {s 0.4 1}
    tree { s 0.96 y 0.33 r -1.4}
}
```

Donne, par exemple :

Cette étape est plutôt simple à comprendre et à effectuer. Notez qu'on utilise le hasard et qu'à chaque *render* nous obtenons un tronc différent.

3.7.3 Étape 3 : l'arbre (Tronc et feuille)

Nous en arrivons déjà à un arbre, et il ne nous faut pour cela qu'ajouter une simple règle au code précédent.

```
startshape tree

rule tree {
    CIRCLE {s 0.4 1}
    tree { s 0.96 y 0.33 r 1.4}
}

rule tree {
    CIRCLE {s 0.4 1}
    tree { s 0.96 y 0.33 r -1.4}
}

rule tree 0.1 {
    tree{r 10}
    tree { r -30 s 0.8}
}
```

Donne, par exemple :



Nous avons-là quelque chose de déjà concret. Mais comment se fait-il que nous obtenons déjà des feuilles alors que nous n'avons ajouté qu'une seule règle ? Je parle bien sûr de la règle *tree* devant laquelle nous avons inséré la valeur 0.1. Il y a donc moins de chance que l'ordinateur choisisse cette règle-là plutôt que les autres lorsqu'on fait appel à la règle *tree*.

Observez bien la règle *tree* que nous avons ajoutée ; elle contient deux instructions. D'abord, on appelle la règle *tree* avec l'argument *r 10*. En bref, on va créer un nouveau tronc qui sera dirigé vers la gauche (à cause de l'angle de 10 degré !). Très bien, mais comment se fait-il que sur l'image obtenue, les nouveaux troncs que l'on crée et qui forment les branches de l'arbre sont plus petits que le tronc principale ? C'est en fait parce que lorsqu'on appelle la règle *tree*, on a pris le soin de fournir un argument *size* qui réduit la taille des troncs créés. C'est pour cela qu'on crée des troncs qui sont de plus en plus petits. Quant à la règle suivante, elle appelle aussi la règle *tree*, avec cette fois-ci deux arguments : un pour l'angle (-30, c'est une valeur que j'ai obtenue à la suite de divers essais, et au final c'est celle qui va le mieux, mais il n'est pas important de comprendre exactement pourquoi j'ai pris cette valeur exactement, retenez juste le fonctionnement principal) et l'autre pour la taille (*s 0.8*, comme je l'ai dit plus haut, on va toujours créer des troncs de plus en plus petits. Pourquoi *0.8* et pas *0.96*, comme dans les deux autres règles *tree* ? Parce que c'est la valeur qui accompagne le mieux cet argument, mais encore une fois, l'important c'est de comprendre le fonctionnement général du code).

Pour résumer le mécanisme de notre programme : on crée un tronc principale. Ensuite, on crée des troncs toujours plus petit, à gauche et à droite.

Au final on ne fait que créer des troncs, mais ils deviennent si petits qu'ils finissent par former les feuilles de l'arbre.

3.7.4 Étape 4 : l'arbre complet

Dans cette étape, nous allons ajouter la même règle que celle qui a été ajoutée lors de l'étape 3, mais cette fois nous inverserons les valeurs que nous allons mettre avec les arguments. Du coup, nous obtiendrons un arbre qui peut aussi bien être feuillu à gauche qu'à droite.

```
startshape tree

rule tree {
    CIRCLE {s 0.4 1}
    tree { s 0.96 y 0.33 r 1.4}
}

rule tree {
    CIRCLE {s 0.4 1}
    tree { s 0.96 y 0.33 r -1.4}
}

rule tree 0.1 {
    tree { r 10 }
    tree { r -30 s 0.8}
}

rule tree 0.1 {
    tree { r -10 }
```

```
tree { r 30 s 0.8}
}
```

Donne, par exemple :



Nous avons maintenant un arbre complet, il ne manque plus que la couleur.

Pour la petite analyse du code, cette fois-ci on a autant de chance d'avoir des feuilles à la gauche qu'à la droite de l'arbre.

Je vais maintenant vous expliquer comment varie l'arbre en fonction des valeurs de certains arguments. Vous allez donc enfin comprendre pourquoi j'ai pris ces valeurs.

Dans les premières règles *tree* du code ci-dessus, observez la ligne *tree {s 0.96 y 0.33 r 1.4}* :

L'argument *size* contrôle le niveau de feuillage de l'arbre. Avec une valeur de 91, nous nous retrouverons avec un arbre d'hiver, mais avec 99 comme valeur, nous obtiendrons un arbre très feuillu.

L'argument *y* contrôle l'espacement entre les ovales qu'on crée. S'il y a trop d'espacements, nous apercevrons des « trous » dans l'arbre.

Pour finir, l'argument *r* contrôle l'inclinaison des feuilles. Plus la valeur est élevée et plus les feuilles bougent vers les côtés, et inversement.

Bon, il nous faut maintenant ajouter de la couleur ! Passons à l'étape 5.

3.7.5 Étape 5 : l'arbre en couleur

Pour cette étape, il m'a fallu un peu de tâtonnement pour obtenir le vert des feuilles. Afin d'obtenir un tronc brun, rien de plus simple : dans les instructions *tree*, celles où les instructions *CIRCLE{}* sont présentes (avec des arguments), il suffit d'ajouter trois arguments : *sat*, *h* (raccourci de *hue*) et *b*(pour *brightness*). Ensuite, à l'aide de l'outil « Color calculator », on cherche une couleur brune, puis on insère les valeurs correspondantes dans notre instruction *CIRCLE{}*.

C'est ce que j'ai fait pour obtenir la couleur brune du tronc.

```

startshape tree

rule tree {
    CIRCLE {s 0.4 1 h 1.49 sat 0.8558 b 0.4641}
    tree { s 0.96 y 0.33 r 1.4 h 0.75}
}

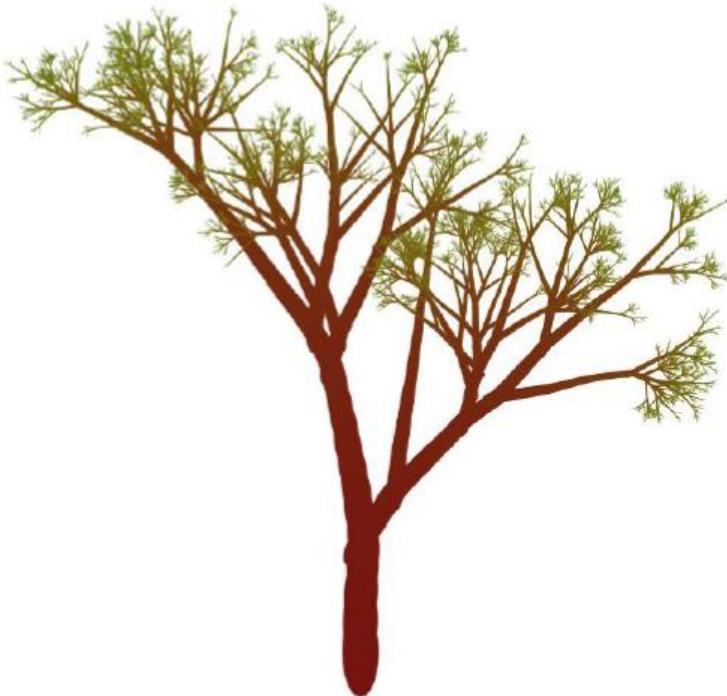
rule tree {
    CIRCLE {s 0.4 1 h 1.49 sat 0.8558 b 0.4641}
    tree { s 0.96 y 0.33 r -1.4 h 0.75}
}

rule tree 0.1 {
    tree { r 10 }
    tree { r -30 s 0.8}
}

rule tree 0.1 {
    tree { r -10 }
    tree { r 30 s 0.8}
}

```

Donne, par exemple :



en fait), ce qui crée une variation vers une teinte verte. Pour trouver cette valeur-là, il m'a fallu essayer d'autres valeurs pour voir comment la couleur brune variait en fonction de la valeur que j'insérais, puis je suis finalement arrivé à la valeur 0.75. Voilà, nous avons notre arbre, et en couleurs !

Analysons le code. Qu'est-ce qui est nouveau ? Premièrement, les arguments *hue*, *saturation* et *brightness* présents dans les instructions *CIRCLE{}}, avec des valeurs spéciales.*

Deuxièmement, c'est l'argument *hue*, accompagné de la valeur 0.75, qui se trouve dans chacune des deux instructions qui suivent les instructions *CIRCLE{}}*.

Cela fait varier la valeur de *hue* au cours de la création des feuilles (des petits troncs

3.7.6 : création d'une forêt

Et si nous essayions de construire une forêt ? Il suffit de créer une règle qui appelle la fonction *tree*, puisque cette dernière crée un arbre simple. Ensuite, cette règle va s'appeler elle-même avec divers arguments.

```
startshape foret

rule tree {
    CIRCLE {s 0.4 l h 1.49 sat 0.8558 b 0.4641}
    tree { s 0.96 y 0.33 r 1.4 h 0.75}
}

rule tree {
    CIRCLE {s 0.4 l h 1.49 sat 0.8558 b 0.4641}
    tree { s 0.96 y 0.33 r -1.4 h 0.75}
}

rule tree 0.1 {
    tree { r 10 }
    tree { r -30 s 0.8}
}

rule tree 0.1 {
    tree { r -10 }
    tree { r 30 s 0.8}
}

rule foret{
    tree{}
    foret{x 1.5}
}

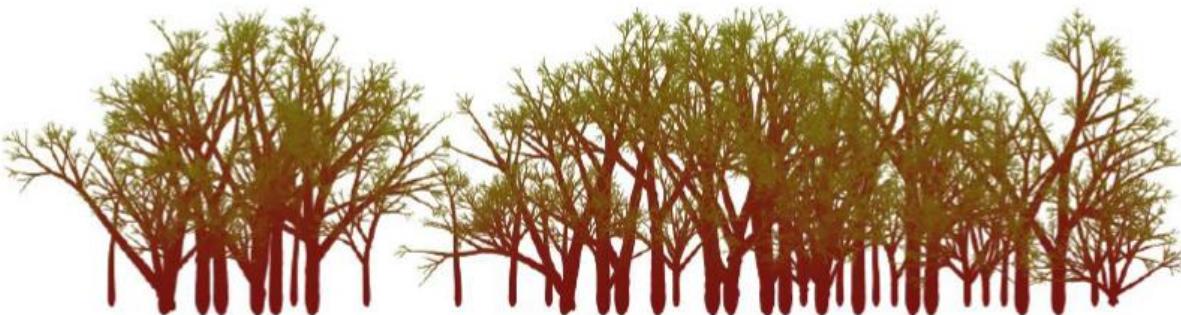
rule foret{
    tree{}
    foret{x -1.5}
}

rule foret{
    tree{s 0.5}
    foret{x 2.5}
}

rule foret{
    tree{s 0.5}
    foret{x -2.5}
}

rule foret 0.1{}
```

Donne :



Et voilà notre forêt ! Avec très peu de lignes de codes, le programme Context Free Art arrive à créer une forêt d'arbres en couleurs !

Analysons le code :

Premièrement, j'ai ajouté 5 règles *foret*. Une de ces règles possède la valeur 0.1, elle a donc moins de chance d'être exécutée par l'ordinateur. Elle ne contient aucune instruction, une fois que l'ordinateur l'exécute il arrête de dessiner.

Pour ce qui est des autres règles *foret*, chacune appelle la règle *tree* (certaines avec un argument *size*, pour avoir des arbres de différentes tailles...). Cela crée un arbre. Puis, dans chacune de ces règles, l'instruction suivante permet d'appeler la règle *forêt* avec un argument *x*, équipé de valeurs variables. C'est ainsi que l'ordinateur crée des arbres un peu partout sur l'axe des *x*.

4. Applications du logiciel

Arrivé à l'aboutissement de ce mode d'emploi, on peut se demander quelles sont les applications possibles du logiciel.

Context Free Art peut être un programme intéressant s'il est utilisé à des fins artistiques. Avec seulement quelques lignes d'instructions, il est possible d'arriver à un résultat surprenant. Que ce soit pour la réalisation de fractales ou de divers dessins, le programme convient parfaitement. On remarque qu'il est souvent utilisé pour réaliser des flocons de neige, des arbres, des fleurs et autres dessins du même genre.

En revanche, on conçoit mal l'idée de l'utiliser pour des branches scientifiques comme les mathématiques. En effet, Context Free Art est un programme qui se base principalement sur la récursivité et le hasard, ce qui le rend imprévisible (l'ajout d'une simple règle peut parfois avoir des conséquences inattendues, à cause de la récursivité). De plus, le fait qu'il n'est pas possible de créer de nouvelles variables rend son utilisation impossible pour certaines applications mathématiques, comme par exemple la représentation graphique de fonctions, à l'inverse d'un programme tel que Mathematica, qui est très pratique pour ce genre de choses. Context Free Art

peut à la limite être utile lorsqu'il s'agit de représenter graphiquement une forme ou un polygone.

Pour ce qui est de son emploi en informatique, Context Free Art ne constitue pas une très bonne approche de la programmation étant donné qu'il n'est pas possible de créer ses propres variables. En revanche, le logiciel peut être intéressant lorsqu'il s'agit d'inculquer une méthode de programmation originale, grâce à son utilisation fréquente de la récursivité. Il peut-être par exemple enrichissant de comprendre la façon dont le programme produit une image puis éventuellement de retranscrire cette manière de faire dans d'autres langages de programmations.

En résumé, une utilisation en classe de Context Free Art est intéressante, s'il s'agit d'utiliser le programme pour des branches artistiques comme le dessin et non pas pour des branches scientifiques comme les mathématiques. Il peut toutefois être intéressant de l'intégrer en informatique, mais uniquement en tant que complément à un autre programme.

5. Conclusion

Au final, Context Free Art se révèle être un programme assez simple d'utilisation lorsqu'il s'agit d'effectuer des dessins basiques. Les instructions de bases dont il se sert ne sont pas très difficiles à comprendre.

Cependant, à mesure que l'on pousse le programme à réaliser des dessins et des structures plus complexes, il devient plus ambigu de trouver la meilleure manière de les réaliser et il est d'autant plus compliqué de comprendre comment l'ordinateur procède afin d'obtenir l'image souhaitée.

Ce qui est surtout intéressant dans ce logiciel, c'est l'importance de la récursivité. Il est surprenant de constater qu'en ajoutant seulement une ou deux règles, l'image que l'on obtient peut radicalement changer.

D'autre part, le programme parvient à dessiner des images complexes avec souvent peu d'écritures. C'est encore une fois un des effets de la récursivité.

En revanche, la récursivité engendre aussi des images dont le code n'est pas facile à comprendre pour un utilisateur autre que le créateur même du code, même avec l'ajout de commentaires.

Context Free Art est donc un programme très intéressant pour une utilisation artistique. Dans ce programme, il existe un nombre inimaginable de façon de créer une certaine forme. Il suffit d'observer quelques créations mises en lignes dans la galerie du site officiel pour constater qu'il y a plus de 50 dessins d'arbres mis en ligne, mais que tous sont différents et qu'ils ont tous été construits de manières différentes.

6. Médiographie

Site internet :

<http://www.magicandlove.com/blog/tag/tutorial/> (Date de consultation : Mars 2011)

<http://matt-pace.tripod.com/index.html> (Date de consultation : Septembre 2011)

<http://blog.loonie.fr/2009/09/13/context-free-art-grammar-introduction/#more-375>
(Date de consultation : Mars 2011)

<http://www.contextfreeart.org/mediawiki/index.php/Tutorials> (Date de consultation : De mars 2011 jusqu'à Décembre 2011).

Je me suis parfois inspiré de certaines images de la galerie du site officiel
<http://www.contextfreeart.org>.

Remarque : La section « documentation » du site officiel
<http://www.contextfreeart.org> ne m'a que très peu servie. Cette page a été modifiée depuis l'arrivée de la version beta 3 du logiciel, je ne peux donc plus retrouver l'ancienne page qui ne concernait que la version 2.2.2. Le lien cité plus haut redirigeant vers plusieurs tutoriaux ne semble pas quant à lui avoir été modifié suite à l'arrivée de la version beta 3.

7. Déclaration

Je déclare par la présente que j'ai réalisé ce travail de manière autonome et que je n'ai utilisé aucun autre moyen que ceux indiqués dans le texte. Tous les passages inspirés ou cités d'autres auteur-es sont dûment mentionnés comme tels. Je suis conscient que de fausses déclarations peuvent conduire le Lycée cantonal à déclarer le travail non recevable et m'exclure de ce fait à la session d'examens à laquelle je suis inscrit.

Ce travail reflète mes opinions et n'engage que moi-même, non pas le professeur responsable de mon travail ou l'expert qui m'a accompagné dans cette recherche.

Lieu et date : Signature :

8. Table des matières

1. INTRODUCTION	4
2. MODE D'EMPLOI DU PROGRAMME.....	4
2.1 TÉLÉCHARGER ET INSTALLER CONTEXT FREE ART.....	4
2.2 FONCTIONNALITÉS DE BASE DU PROGRAMME	5
2.2.1 Les onglets	6
2.2.2 La console du programme et l'outil « color calculator ».....	7
2.2.3 Les boutons et les sous-onglets	8
3. MODE D'EMPLOI DE LA GRAMMAIRE.....	9
3.1 LES FORMES.....	9
3.1.1 Création de formes de base	9
3.1.3 Étoiles	12
3.1.4 En résumé des trois derniers points :	12
3.2 SYSTÈME DE RÈGLES	13
3.2.1 Pousser un peu plus loin le système de règle	13
3.3 FRACTALE.....	14
3.3.1 Triangle de Sierpiński et son dérivé	14
3.3.2 Mélange de fractale et de système de règle	18
3.3.3 Obtenir un tapis de Sierpiński avec des cercles.....	19
3.4 UTILISATION DU HASARD	21
3.4.1 Élaboration d'un chemin de cercle.....	21
3.4.2 Rotation d'un angle aléatoire	23
3.5 UTILISATION DES CHEMINS	28
3.5.1 création de lignes et de polygones	28
3.5.2 Expérience et contrôle de l'épaisseur du trait.....	31
3.5.3 Arc de cercles	32
3.5.4 Synthèse des derniers points.....	34
3.6 UTILISATION DE COULEURS	35
3.6.1 Quelques couleurs de bases :.....	36
3.6.2 Utilisation d'un dégradé :	37
3.6.3 Une autre utilisation de la couleur.....	38
3.7 PROJET FINALE : CONSTRUCTION D'UN ARBRE.....	39
3.7.1 Étape 1 : Le tronc	40
3.7.2 Étape 2 : le tronc qui penche des deux côtés	40
3.7.3 Étape 3 : l'arbre (Tronc et feuille)	41
3.7.4 Étape 4 : l'arbre complet	42
3.7.5 Étape 5 : l'arbre en couleur.....	43
3.7.6 : création d'une forêt	45
4. APPLICATIONS DU LOGICIEL	46
5. CONCLUSION	47
6. MÉDIAGRAPHIE	48
SITE INTERNET :	48
7. DÉCLARATION	48

