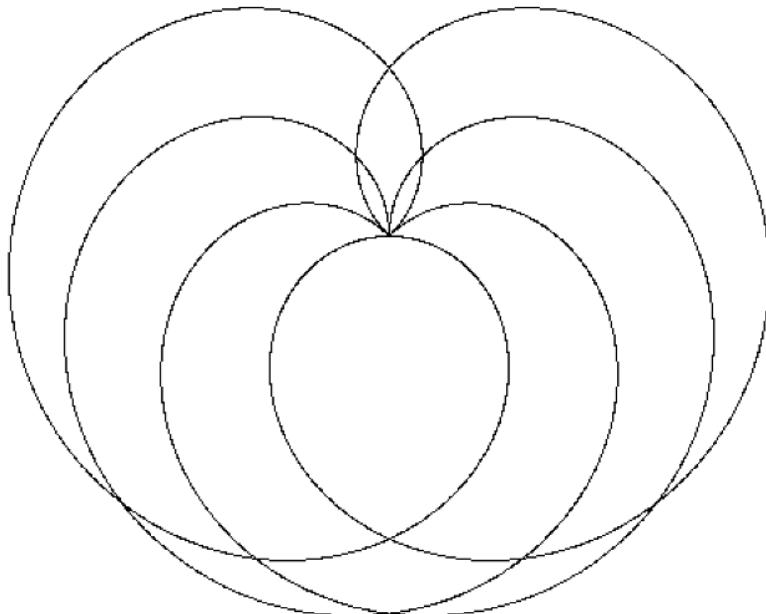


Meetup CreativeCodeParis

<https://www.meetup.com/fr-FR/CreativeCodeParis/>

Premiers pas avec P5.js



Manuel pour aider les
Bourdons Codeurs à
prendre leur envol

Version 1.0

Table des matières

1.	Introduction	6
1.1	Préambule	6
1.2	Se lancer avec P5.js.....	8
1.3	Quels outils utiliser	9
1.3.1	Travailler avec OpenProcessing	11
1.3.2	Travailler avec Codepen.....	12
1.3.3	Travailler avec Live.codecircle.com.....	15
1.3.4	Travailler avec des éditeurs de code	17
2.	Maîtriser les bases de P5	19
2.1	Squelette de sketch.....	19
2.2	Premiers sketchs pour dessiner à main levée	22
2.3	Approfondissement sur la notion de variable	29
2.4	La fonction Keypressed est votre amie.....	35
2.5	Premier effet d'animation.....	37
2.6	Créer sa propre fonction « line »	40
2.7	Un peu de communication avec l'utilisateur	44
2.8	Des algorithmes qui dessinent sous nos yeux.....	48
2.8.1	Les primitives proposées par P5.....	49
2.8.2	Tracé de cercle – algo n° 1 (avec racine carrée)	51
2.8.3	Tracé de cercle – algo n°2 (avec sinus & cosinus)	56
2.8.4	Compas avec effet de traîne	60
2.8.5	Cercle en folie.....	66
2.8.6	Polygones.....	68
3.	Partir à l'aventure	70
3.1	Courbes polaires.....	70
3.1.1	Abeilles polaires.....	70
3.1.2	Abeilles survitaminées	81
3.2	Curve Stitching.....	88
3.2.1	Courbe de poursuite (« curve of pursuit »).....	90
3.2.2	Tractrix	94
3.2.3	Rose mystique.....	97
3.2.4	Epicycloïde.....	99
3.2.5	Huit paraboles dans un carré, avec P5.dom.....	103
4.	Conclusion	110
5.	Références bibliographiques	111
A.	Annexe	113
A.1	Window.isNaN ou Number.isNaN.....	113

A.2 Polygone animé	115
A.3 Les tableaux en JS.....	116
A.4 Les fonctions en JS.....	120
A.5 Expérimentations	124

L'auteur :

Je m'appelle Grégory Jarrige. Je suis développeur et formateur indépendant. Je développe des applications depuis plus de 30 ans. L'arrivée du HTML5 (vers 2009-2010) et de ses nombreuses APIs – à commencer par Canvas et Webaudio - a été pour moi une véritable bouffée d'oxygène, et m'a donné envie de me replonger dans les aspects ludiques de la programmation, que j'avais délaissées au fil du temps. Ma découverte de projets comme Processing et P5js est plus récente, et j'ai eu envie de rencontrer d'autres développeurs utilisant ces projets. Au début de l'année 2016, il n'existe pas de meetup véritablement consacré au « Creative Coding » (en tout cas pas à Paris), cela m'a conduit à fonder en mai 2016 le meetup CreativeCodeParis (qui s'appelait un peu différemment au départ). J'ai été rejoint par une petite bande de passionnés, développeurs, graphistes et musiciens, et c'est une belle aventure qui se poursuit en 2017, avec des rendez-vous réguliers (généralement mensuels).

<https://www.meetup.com/fr-FR/CreativeCodeParis/>

Blog : <http://gregphlab.com>

Github : <https://github.com/gregja>

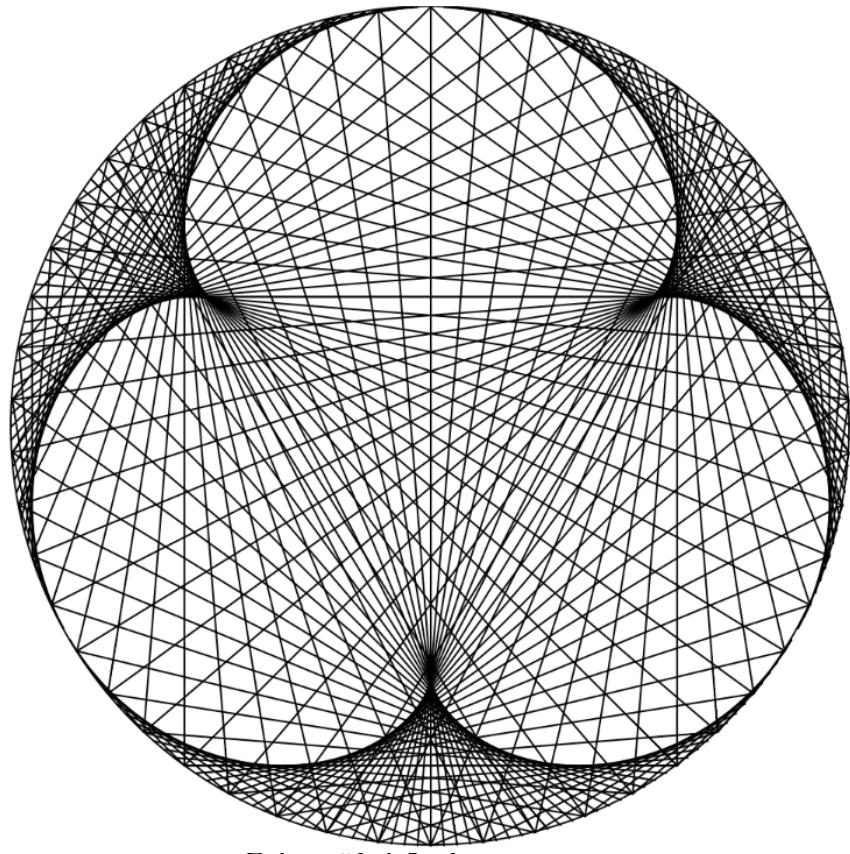
Twitter : https://twitter.com/greg_devlab

Selecteurs :

Carole Lambert et Gregor Schneider sont deux développeurs passionnés par le projet P5, qui pensent – comme votre serviteur – que P5 est un merveilleux outil pour l'apprentissage de la programmation. Carole et Gregor ont généreusement donné de leur temps pour relire attentivement ce support, et me signaler tous les points nécessitant des corrections ou des clarifications. Je souhaite ici leur exprimer toute ma reconnaissance.

J'en profite pour signaler que Carole Lambert organise des ateliers d'initiation à la programmation pour les enfants. Pour de plus amples renseignements à ce sujet :

<http://www.code14.fr/>



Epicycloïde à 3 rebroussements

1. Introduction

1.1 Préambule

Lors des différentes rencontres organisées dans le cadre du meetup « Creative Code Paris », j'ai constaté une attente très forte de la part de nombreuses personnes, qui souhaitaient être accompagnées dans la prise en main de projets tels que P5.js et Processing.

J'ai pris dans le même temps conscience de la difficulté inhérente à l'organisation d'ateliers d'initiation, sachant que les personnes présentes aux meetups ont des niveaux de connaissance en programmation extrêmement hétérogènes, et bien sûr des attentes très différentes.

Il existe beaucoup de bons livres et de bons tutoriels sur Processing et P5, mais la grande majorité sont en anglais, ce qui complique la tâche des personnes débutant en programmation. Il n'est en effet pas facile de s'approprier des concepts nouveaux quand ils sont expliqués dans une langue que l'on ne maîtrise pas parfaitement.

Durant la rédaction de ce support, j'ai quand même découvert deux très bons tutoriels en français, que j'ai indiqués au tout début du chapitre « références ». Je pense que vous prendrez plaisir à les étudier, ils sont très complémentaires du support que vous êtes en train de lire.

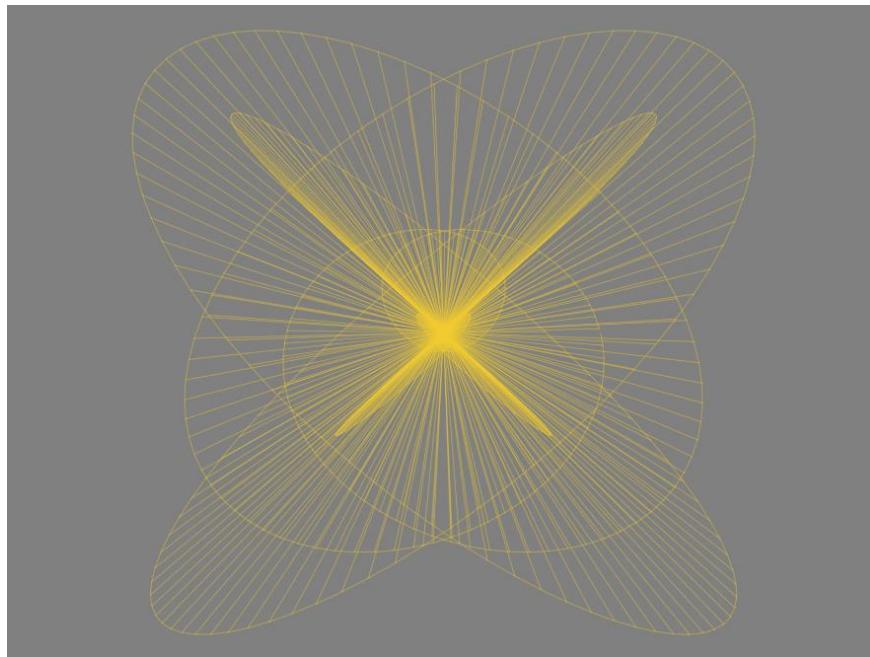
Lors de la rédaction de ce support, je me suis efforcé d'apporter, à la fois des techniques de programmation, et quelques outils méthodologiques, pour aider les lecteurs à comprendre ce qui se passe sous le capot des navigateurs. Les quelques algorithmes que j'ai écrits en français ont pour but d'aider le lecteur à prendre un peu de hauteur par rapport au code concerné.

Voici quelques unes des questions auxquelles je me suis efforcé de répondre :

- Comment se lancer, avec quels outils ?
- Comment créer des formes simples et complexes via l'écriture de sketchs ?
- Comment animer ces formes, faire en sorte que leur tracé apparaisse de manière progressive à l'écran ?

Pour essayer de répondre à ces questions, j'ai privilégié des modèles graphiques que l'on peut qualifier de « vintages ». Ils ne sont que rarement abordés dans les ouvrages d'initiation publiés aujourd'hui. C'est dommage, car ces modèles graphiques permettent de manipuler des notions algorithmiques et géométriques intéressantes, de s'initier en douceur et de manière ludique, avec un résultat concret et immédiat.

Voici un exemple d'image que vous apprendrez à créer dans le cadre de ce tutoriel :



... et en plus vous apprendrez à l'animer 😊.

Ce tutoriel est pour vous, alors n'hésitez pas à m'écrire pour me suggérer des corrections et des améliorations, me dire sur quels points vous avez buté, quels points ont besoin d'être clarifiés.

1.2 Se lancer avec P5.js

J'ai fait le choix dans ce tutoriel de ne pas trop parler de Processing, et de m'intéresser plutôt à P5.js (dans la suite de ce document je l'appellerai tout simplement P5).

Processing est écrit avec le langage Java et est destiné à faire fonctionner des scripts écrits eux aussi en Java.

Bâti sur les fondations de Processing, P5 est écrit en Javascript (JS) et permet d'écrire des sketchs en pur code Javascript.

Un sketch, dans le jargon de P5 et Processing, c'est un petit programme destiné à générer des images et éventuellement des animations.

Javascript est l'un des meilleurs langages pour s'initier à la programmation de graphisme et d'animation. Langage très souple et très polyvalent, on peut l'utiliser partout, et en particulier dans tous les navigateurs. Associé à la norme HTML5 et à ses nombreuses APIs (dont Canvas, SVG et WebAudio), Javascript constitue un formidable terrain d'expérimentation pour le « creative coding ».

P5 peut être vu comme un moyen de simplifier l'utilisation de l'API Canvas, sur laquelle il s'appuie pour générer graphiques et animations au sein des navigateurs web. Mais P5 est plus que cela, car il apporte un grand nombre de fonctionnalités qui permettent de se concentrer sur l'histoire (graphique) que l'on souhaite raconter. Nous en découvrirons quelques unes au fil de l'eau.

Pour autant, se lancer sur ce terrain de jeu, quand on débute en programmation, n'est pas chose facile, cela peut même être intimidant et décourageant. Aussi je vous propose dans ce tutoriel de démarrer avec quelques outils faciles à prendre en main, et de faire un petit bout de chemin ensemble. J'espère qu'au bout de ce chemin, vous vous sentirez suffisamment en confiance pour vous lancer sereinement et en toute autonomie. En tout cas, c'est mon objectif.

Attention, le document que vous lisez n'est pas un support de cours exhaustif sur P5, pas plus que sur Javascript. Si c'était le cas, il ferait très certainement plus de 800 pages. Mais je vous donnerai des conseils de lecture, pour que vous puissiez approfondir certains sujets clés.

1.3 Quels outils utiliser

Il est intéressant de noter que la fondation Processing travaille activement à l'écriture d'un environnement de développement qui permettra de développer des sketchs pour P5 en ligne, c'est-à-dire directement à l'intérieur d'un navigateur. Cet outil devrait être disponible dans le courant de l'année 2017.

En attendant la disponibilité de cet outil, il existe déjà plusieurs manières de travailler, qui conviendront à différents usages et à différents types de développeurs (de niveau débutant et avancé).

La première manière consiste à développer son code en local, en utilisant un serveur web local. Ce serveur web local peut être piloté par un environnement PHP, Python, Ruby ou NodeJS...

Si vous êtes développeur de niveau avancé, vous aurez certainement une préférence pour l'un ou l'autre des environnements précités, et dans ce cas vous n'avez pas besoin de moi pour créer une page web, y insérer P5.js et votre premier sketch. Vous êtes certainement désireux d'avancer vite vers l'étude de sketchs concrets.

Si vous êtes développeur débutant, vous devez être bien embarrassé par cette profusion d'outils. Lequel choisir ? Aucun, ou plutôt aucun de ceux que je viens d'évoquer. Je vous propose plutôt d'utiliser l'une des trois solutions suivantes :

- Openprocessing.org
- Codepen.io
- Live.codecircle.com

Openprocessing.org est un site internet spécialisé dans l'hébergement de sketchs Processing et P5. C'est une sorte de labo ouvert à tous, vous pouvez y observer le travail des autres développeurs, leur piquer des idées, explorer, expérimenter, c'est un outil vraiment cool.

Codepen.io offre un espace de travail et d'échange assez similaire à Openprocessing, à la différence que Codepen n'est pas dédié à P5 et Processing (il est beaucoup plus généraliste). Mais nous verrons qu'il est très facile de le configurer pour travailler avec P5.

Live.codecircle.com, c'est un peu le challenger, je l'ai découvert fin mars 2017, pendant la rédaction de ce support. Développé par la « Goldsmiths University of London » pour ses étudiants, cette solution est ouverte depuis peu au grand public, et on peut y créer gratuitement un compte et créer ses propres sketchs, ou tous types de projets Javascript. Le support de P5 y est intégré d'office. L'outil est simple et agréable à utiliser, comme nous le verrons dans un instant.

Vous allez peut être me dire qu'un choix de trois solutions, c'est beaucoup, et que vous aimerez bien être un peu plus orienté. Honnêtement, j'aime bien ces trois solutions, et je trouve qu'elles ont toutes des points forts... alors voilà comment je vois les choses :

- Openprocessing.org est une solution dédiée à Processing et à P5. C'est l'endroit idéal pour découvrir des projets dédiés à Processing et P5, et échanger avec les « creative coders » qui se concentrent à ces outils. C'est un passage obligé, pour voir ce qui se fait, et montrer aussi aux autres ce que vous créez avec P5 (ou Processing).
- Codepen.io est une solution généraliste, très appréciée des développeurs pros qui s'en servent à la fois pour du prototypage, et comme vitrine de leur travail. Une nouveauté très récente de Codepen consiste à pouvoir définir de véritables projets (avec plusieurs fichiers et répertoires) comme dans un environnement de développement local. Codepen est donc une solution plutôt orientée pour les professionnels, mais les développeurs amateurs peuvent prendre plaisir à travailler avec. La version gratuite est fonctionnelle, mais pour bénéficier d'un maximum de fonctionnalités, il est recommandé de souscrire à un abonnement (peu onéreux). A noter qu'on trouve sur Codepen beaucoup de très bons « pens » utilisant les APIs Canvas et Webaudio, la newsletter de codepen est d'ailleurs excellente.
- Live.codecircle.com est une solution généraliste, développée par une université prestigieuse, la Goldsmith University of London. Cette plateforme est gratuite, conviviale, et supporte nativement P5. Elle est moins connue que Codepen et Openprocessing, donc ce n'est peut-être pas le meilleur endroit pour exposer votre travail, mais c'est un endroit où l'on peut prendre plaisir à coder, particulièrement quand on débute. Parlez en autour de vous, c'est vraiment une chouette plateforme.

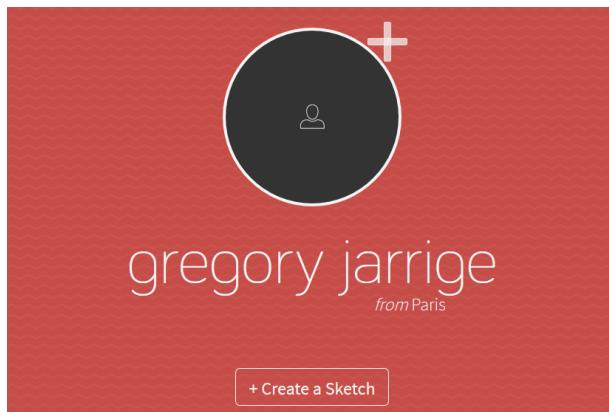
1.3.1 Travailler avec OpenProcessing

En consultant la page d'accueil d'Openprocessing, vous verrez de nombreux sketchs proposés par la communauté des développeurs. Je vous invite à en sélectionner un, n'importe lequel, pour vous faire une idée sur la manière dont Openprocessing fonctionne.

Vous verrez en haut de l'écran une barre constituée de 3 icônes : 

L'icône à gauche permet de prendre connaissance des informations générales du sketch sélectionné. L'icône du milieu permet d'exécuter le sketch. L'icône de droite permet de consulter le code source du sketch. C'est avec ce dernier mode que vous créerez vos propres sketchs.

Pour créer vos propres sketchs, vous devez créer un compte (c'est gratuit). A partir de là, vous pouvez entrer dans le vif du sujet :



Lorsque vous créez un nouveau sketch, Openprocessing vous propose d'office un petit sketch que vous pouvez utiliser comme squelette pour développer vos propres projets :



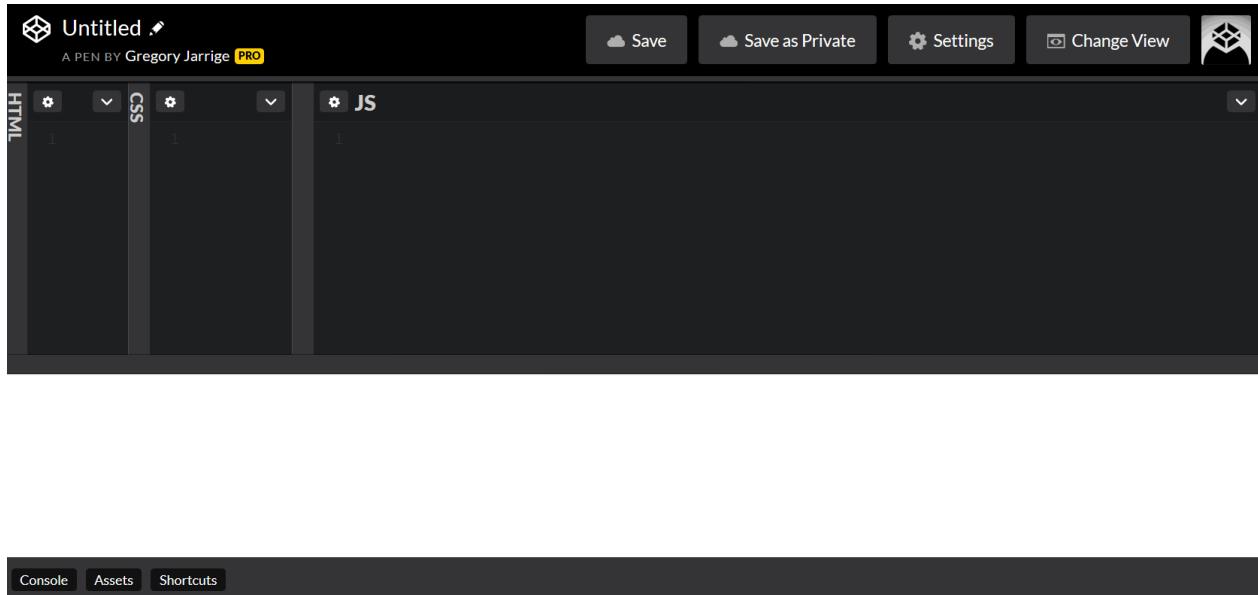
```
function setup() {
  createCanvas(windowWidth, windowHeight);
  background(100);
}

function draw() {
  ellipse(mouseX, mouseY, 20, 20);
}
```

Quand votre sketch est prêt, pensez à cliquer sur le bouton « save » pour ne pas perdre votre travail. Pensez aussi à modifier le titre de votre sketch pour aider les autres développeurs à déterminer si votre sketch les intéresse.

1.3.2 Travailler avec Codepen

Avec Codepen, le principe est assez similaire à Openprocessing, si ce n'est que l'objectif prioritaire de Codepen est d'être utilisé comme un outil de prototypage pour construire des pages webs, c'est la raison pour laquelle vous verrez un écran de travail contenant plusieurs parties (HTML, CSS et JS) comme dans l'exemple suivant :



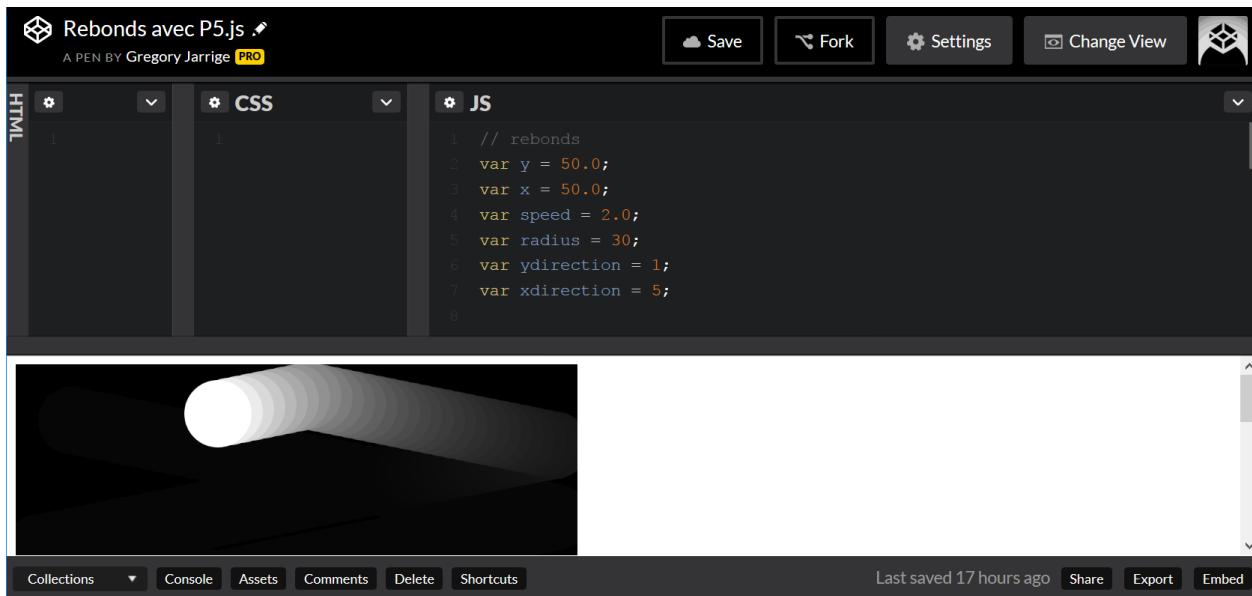
L'écran ci-dessus, c'est l'espace de création d'un « pen ». Un « pen » c'est un prototype destiné à tester, montrer, démontrer. Tout ce qui vous passe par la tête, et tout ce qui peut être écrit en HTML, CSS et/ou JS peut être placé dans un « pen ».

Je vous recommande de prendre un moment pour lire la présentation officielle de Codepen, elle donne un bel aperçu des possibilités de l'outil :

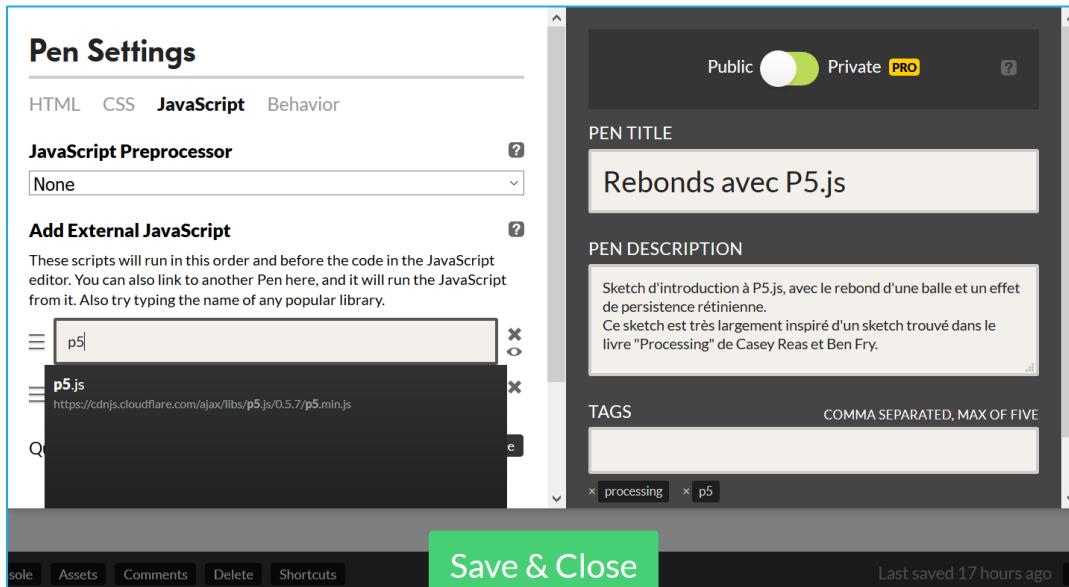
<https://codepen.io/hello/>

Dans un premier temps, vous travaillerez beaucoup plus sur la partie du code dédiée au JS, vous n'aurez pas trop besoin du HTML et du CSS, aussi je vous recommande de réduire la taille des cadres HTML et CSS comme je l'ai fait dans la copie d'écran ci-dessus.

Quand votre sketch sera en mesure de fonctionner, vous verrez apparaître le résultat dans la seconde moitié de l'écran, comme dans l'exemple suivant :



Pour que votre sketch P5 fonctionne, vous devez configurer Codepen de manière à ce qu'il importe le projet P5.js dans votre « Pen ». Pour ce faire, cliquez sur le bouton « Settings » (en haut à droite de l'écran) puis dans l'onglet « Javascript » dans l'écran ci-dessous :



Vous devez indiquer à Codepen que vous souhaitez travailler avec le projet P5, aussi saisissez « p5 » sous « Add external Javascript ».

Codepen recherche le code source de P5 dans un CDN (Content Delivery Network), qui s'appelle « cdnjs ». C'est en fait une sorte d'annuaire dédié aux projets Javascript.

En mars 2017 (date de rédaction de ce document), la version la plus récente de P5.js est la 0.5.7. C'est la raison pour laquelle Codepen me propose cette version, que je sélectionne :

<https://cdnjs.cloudflare.com/ajax/libs/p5.js/0.5.7/p5.min.js>

Cliquez sur le bouton « Save & Close », c'est prêt, vous pouvez commencer à coder votre premier sketch.

Un truc qui est cool dans Codepen, c'est la possibilité de « fork » ses propres « Pen » ainsi que ceux des autres. Derrière le terme de « fork », il y a l'idée de copier pour explorer d'autres voies, de devenir indépendant du projet initial.

Dans le cas de vos propres « Pens » (ou sketches), si vous avez mis au point un « Pen » sympa, mais que vous envisagez de le modifier, pensez à le « fork », histoire d'avoir une version « avant » et une version « après ». Si vos dernières modifications ne vous plaisent pas, vous pourrez supprimer ce « Pen », sachant que le « Pen » d'origine est toujours là.

Dans le cas des « Pens » d'autres développeurs, vous pouvez là encore les « fork » et du coup avoir une copie sous votre propre profil utilisateur. Vous pouvez ainsi décortiquer le code des copains, le modifier si l'envie vous en prend. Si vous faites ça, pensez à mettre le « Pen » dupliqué en visibilité « privée » (mode « Private »), pour ne pas « polluer » l'espace public avec des projets redondants, qui de surcroît ne sont pas de vous.

1.3.3 Travailler avec Live.codecircle.com

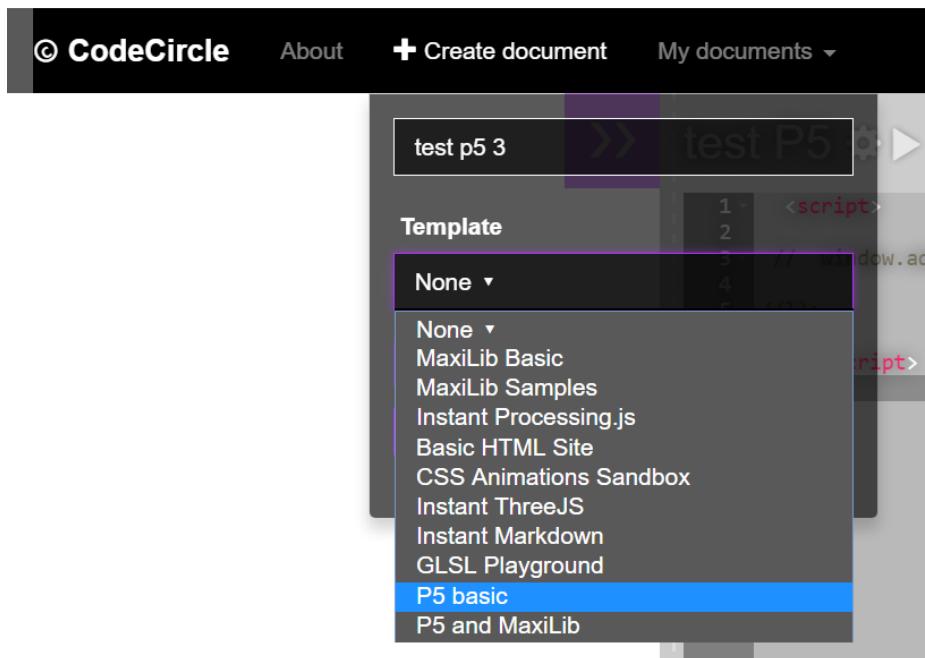
Je pensais au départ à deux solutions (Openprocessing et Codepen), mais j'ai découvert l'existence de Live.codecircle.com tout récemment en suivant l'excellent cours du Dr Matthew Yee King (qui est développeur, musicien et professeur) sur l'API Webaudio :

<https://www.futurelearn.com/courses/electronic-music-tools/>

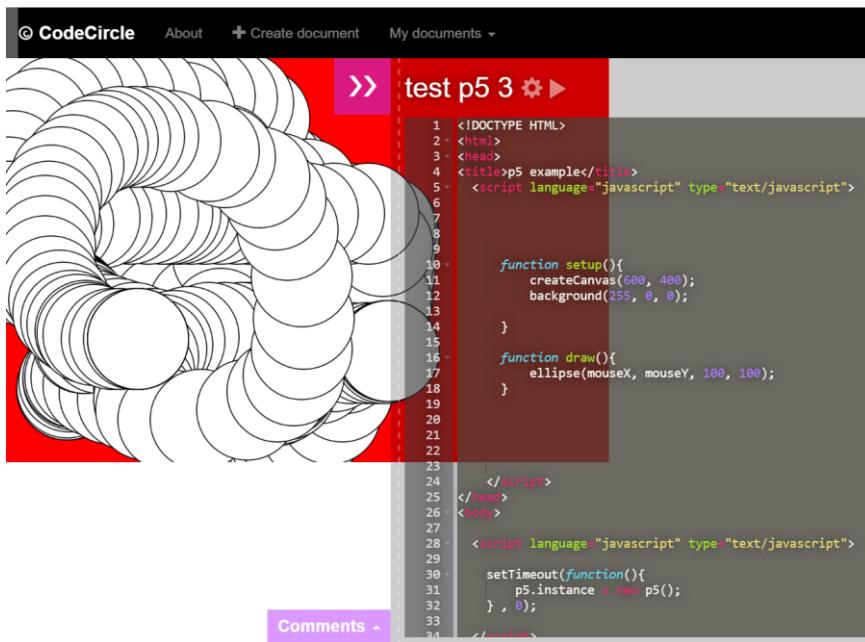
Live.codecircle.com est un site développé par la Goldsmiths University of London pour ses étudiants. Cette plateforme me semble particulièrement bien adaptée à l'apprentissage du Javascript, et elle supporte nativement P5, alors je me devais de la présenter ici.

La création d'un compte sur Live.codecircle.com est gratuite, et se fait en quelques secondes.

Créer un sketch P5 avec cette solution en ligne est très simple. Après avoir cliqué sur « Create Document », il suffit de choisir « p5 basic » dans l'option « Template » :

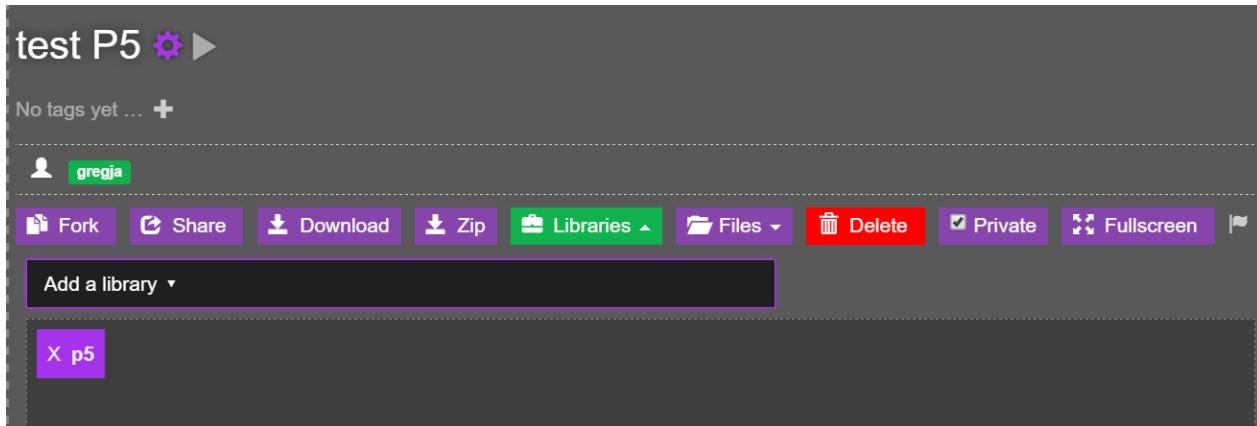


Un modèle de sketch P5 apparaît ensuite, sketch que vous allez pouvoir personnaliser à votre guise :



```
>> test p5 3 >
1  <!DOCTYPE HTML>
2  <html>
3  <head>
4  <title>p5 example</title>
5  <script language="javascript" type="text/javascript">
6
7
8
9
10 <function setup(){
11   createCanvas(600, 400);
12   background(255, 0, 0);
13 }
14
15 <function draw(){
16   ellipse(mouseX, mouseY, 100, 100);
17 }
18
19
20
21
22
23
24 </script>
25 </head>
26 <body>
27
28 <script language="javascript" type="text/javascript">
29
30 <setTimeout(function(){
31   p5.instance = new p5();
32 }, 0);
33 </script>
```

Le paramétrage d'un sketch est très simple, on y accède en cliquant sur l'icône en forme d'engrenage. On peut définir un sketch comme étant privé (private), on peut « fork » ses projets et ceux des autres, on peut aussi ajouter d'autres librairies en plus de P5 (le choix est moins étendu que sur Codepen, mais on peut s'attendre à ce que ça évolue).



1.3.4 Travailler avec des éditeurs de code

Vous aurez peut être envie de saisir le code source de vos sketchs dans des lieux ne fournissant pas de connexion internet. Or, Openprocessing et Codepen sont des outils accessibles uniquement par internet. Il serait dommage que vous vous retrouviez bloqué à cause de cela.

Pour travailler sur votre PC en local, plusieurs éditeurs peuvent faire l'affaire, tels que Notepad++, SublimeText ou Atom. Tous ces outils sont parfaits pour débuter (Sublime Text et Atom sont aussi plébiscités par de nombreux développeurs professionnels). Certains développeurs préfèreront sans doute un environnement de développement intégré tels que Netbeans, Eclipse, Webstorm. Ils sont très bien pour développer des projets Web, mais ces outils sont quelque peu sur-dimensionnés pour des développeurs débutants... enfin c'est à vous de voir 😊.

Quand vous commencez à saisir du code dans un éditeur, ce dernier vous fournira une coloration syntaxique qui facilitera la lecture du code, et pourra vous aider à détecter d'éventuels bugs. Les éditeurs proposent une coloration syntaxique à partir du moment où ils parviennent à identifier le type de code source sur lequel vous travaillez. C'est l'extension du fichier source qui permet à l'éditeur de déterminer le langage principal utilisé au sein du fichier et de proposer une coloration syntaxique en rapport. Si vous ne mettez pas d'extension, ou si vous mettez une extension « .txt » (pour fichier « texte »), alors vous ne bénéficierez d'aucune coloration syntaxique. Si vous mettez une extension « .py » (correspondant au langage Python) alors vous bénéficierez d'une coloration syntaxique adaptée au langage Python. Pour le langage Javascript, l'extension est « .js » (par exemple : sketch_01.js).

Exemples de coloration syntaxique, pour le même extrait d'un sketch P5 :

Dans Codepen	Dans Notepad++
<pre>1 // rebonds 2 var y = 50.0; 3 var x = 50.0; 4 var speed = 2.0; 5 var radius = 30; 6 var ydirection = 1; 7 var xdirection = 5; 8 9 function setup() { 10 createCanvas(500, 500); 11 background(0); // fond noir 12 smooth(); // rendu amélioré 13 noStroke(); // pas de contour 14 ellipseMode(RADIUS); // le centre est la référence de l'ellipse 15 }</pre>	<pre>1 // rebonds 2 var y = 50.0; 3 var x = 50.0; 4 var speed = 2.0; 5 var radius = 30; 6 var ydirection = 1; 7 var xdirection = 5; 8 9 function setup() { 10 createCanvas(500, 500); 11 background(0); // fond noir 12 smooth(); // rendu amélioré 13 noStroke(); // pas de contour 14 ellipseMode(RADIUS); // le centre 15 }</pre>

2. Maîtriser les bases de P5

2.1 Squelette de sketch

Un sketch P5 contient toujours au minimum les 2 fonctions setup et draw que voici :

```
function setup() {  
}  
  
function draw() {  
}
```

Deux fonctions au sens Javascript du terme, ce sont deux blocs de code indépendants, que l'on pourra lancer l'un et l'autre à la demande. Mais on ne le fera pas, en tout cas pas pour ces deux fonctions, car c'est P5 qui va piloter à notre place l'exécution de ces fonctions, en procédant de la façon suivante :

- La fonction « setup » est lancée une seule fois par P5, elle sert à mettre en place l'environnement de travail de notre sketch, comme par exemple la fenêtre d'affichage du sketch (avec sa taille, sa couleur de fond, etc..).
- La fonction « draw » tourne en boucle : sa première exécution s'effectue juste après la fin de l'exécution de la fonction « setup », ensuite elle va être relancée par P5, à un rythme d'environ 60 fois par seconde.

Comme c'est dans la fonction « setup » que nous initialisons notre zone de dessin - notre « canvas » - c'est là que nous placerons la fonction P5 permettant de définir la taille horizontale et verticale du canvas. Il s'agit de la fonction « createCanvas » :

```
function setup() {  
    createCanvas(600, 400); // 600 pixels en largeur, 400 pixels en hauteur  
}  
  
function draw() {  
}
```

Vous avez peut être remarqué que j'ai placé sur la même ligne que la fonction « createCanvas » un peu de code complémentaire commençant par 2 barres obliques //.

Le code qui se trouve à droite des barres obliques ne s'exécutera pas, il s'agit de commentaire. On peut placer ses commentaires un peu comme on veut, et il existe 2 manières de saisir du commentaire. Dans l'exemple ci-dessous, seule la fonction « createCanvas » sera exécutée :

```

// ceci est un commentaire qui ne sera pas exécuté
createCanvas(600, 400) ; // ça aussi c'est un commentaire
// ceci est un autre commentaire qui ne sera pas non plus exécuté
/* ceci est une autre manière de mettre du code en commentaire */
/* ceci est commentaire saisi sur plusieurs lignes, c'est très pratique
quand vous avez beaucoup de choses à raconter
*/

```

Il existe une manière légèrement différente de déclarer un bloc de commentaire multi-lignes, mais je n'en parle pas tout de suite, car à ce stade ce n'est pas important, j'y reviendrai plus tard.

La fonction « draw » a un fonctionnement vraiment particulier. Comme je l'indiquais plus haut, elle est relancée automatiquement par P5 à un rythme d'environ 60 fois par seconde. Cette vitesse d'exécution peut varier en fonction de la puissance de votre terminal (ordinateur, tablette, smartphone), et aussi en fonction de la complexité du code que vous allez placer à l'intérieur de cette fonction « draw ». Si le traitement est complexe, il va mettre un certain temps pour s'exécuter, et le nombre d'exécutions par seconde de la fonction « draw » va baisser. Nous reviendrons sur ce sujet ultérieurement.

A chaque fois qu'un cycle d'exécution de la fonction « draw » se termine, l'image générée pendant ce cycle est envoyée au canvas, qui l'affiche à l'écran. Quand la fonction « draw » est exécutée à un rythme de 60 fois par seconde, ce sont donc 60 images qui sont générées par seconde. Les développeurs anglophones parlent de « frame » plutôt que d'image, c'est la raison pour laquelle vous verrez souvent dans les documentations le terme FPS, pour « frame per second ».

On peut réduire volontairement le nombre de FPS, si l'on souhaite ralentir une animation, on utilisera dans ce cas la fonction « frameRate » fournie par P5. En général on fixera cette valeur à l'intérieur de la fonction « setup », mais rien ne nous empêche de la modifier en cours de route à l'intérieur de la fonction « draw ».

On peut par exemple fixer le nombre de Frames par seconde à 20 :

```

function setup() {
  createCanvas(600, 400) ;
  frameRate(20) ;
}

```

P5 met à notre disposition deux éléments importants :

- la variable « frameCount » qui nous indique le numéro de frame en cours (elle est incrémentée automatiquement par P5 à chaque appel de la fonction « draw »),

- la fonction « noLoop » qui permet de stopper la boucle d'exécution de la fonction « draw ».

Dans l'exemple ci-dessous, dès que la variable « frameCount » est supérieure à 100, on appelle la fonction P5 « noLoop », ce qui a pour effet de stopper la boucle d'exécution de la fonction « draw » :

```
function draw() {  
    // ... (code pour dessiner ici) ...  
    if (frameCount > 100) noLoop();  
}
```

NB : Si vous connaissez un peu Arduino, vous remarquerez la grande similitude avec les fonctions `setup()` et `loop()` d'Arduino.

2.2 Premiers sketchs pour dessiner à main levée

Voici un premier sketch permettant de dessiner à “main levée” en baladant la souris au dessus de la zone d'affichage :

```
function setup() {
    createCanvas(600, 400);
}

function draw() {
    point(mouseX, mouseY);
}
```

Notre sketch crée une fenêtre de dessin (un “canvas”) de 600 pixels de large, sur 400 pixels de haut.

A l'intérieur de la fonction “draw”, nous faisons appel à la fonction “point” qui est fournie par P5 et dont le nom est plutôt explicite. Nous transmettons à cette fonction “point” deux paramètres que sont les variables “mouseX” et “mouseY”. Ces deux variables sont également fournies par P5 :

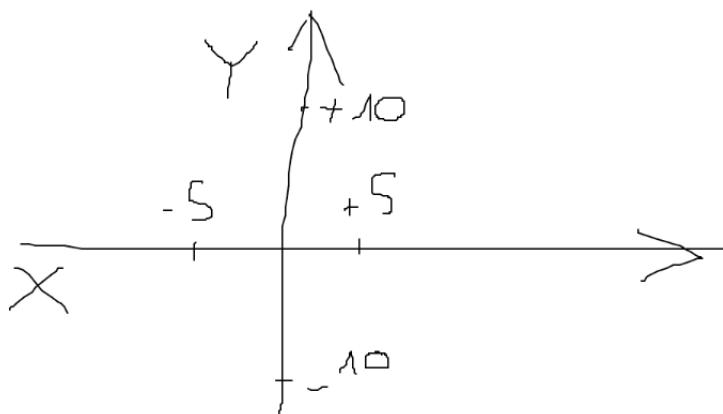
- mouseX contient la position de la souris sur l'axe horizontal (souvent appelé “l'axe des X”, ou encore “l'abscisse”)
- mouseY contient la position de la souris sur l'axe vertical (souvent appelé “l'axe des Y”, ou encore “l'ordonnée”)

Voici un exemple d'image obtenue après avoir baladé la souris sur l'écran pendant quelques secondes.



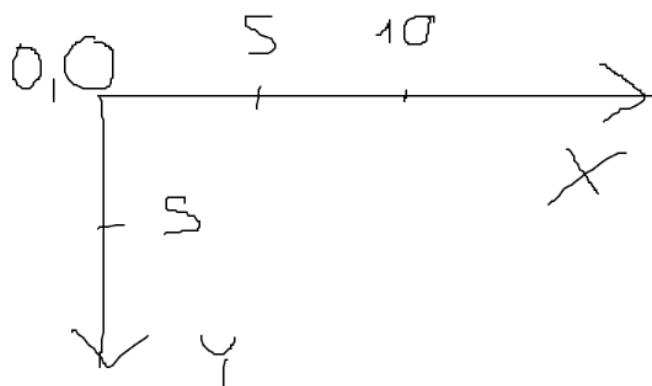
Si vous vous rappelez de vos cours de géométrie, vous devez sans doute vous rappeler que l'on représente le plus souvent les axes X et Y de la façon suivante :

TODO : graphique un peu moche à refaire



Sur l'écran de l'ordinateur, les coordonnées sur l'axe vertical sont inversées. Donc plus notre souris se dirige vers le bas de l'écran et plus la valeur de Y augmente. De plus, le point de coordonnées 0,0 est le coin supérieur gauche de l'écran.

TODO : graphique un peu moche à refaire



OK, c'est cool, mais un peu limité, car le crayon virtuel commence à dessiner dès que l'on déplace la souris... Pas facile de dessiner quelque chose de joli dans ces conditions.

Ajoutons la possibilité d'activer et de désactiver notre crayon virtuel via un clic de souris. La variable “mouseIsPressed”, qui est fournie par P5, va nous aider à résoudre ce problème :

```
function setup() {
    createCanvas(600, 400);
}

function draw() {
    if (mouseIsPressed) {
        point(mouseX, mouseY);
    }
}
```

J'ai maintenant un meilleur contrôle sur mon crayon, je peux commencer à dessiner quelque chose de reconnaissable... plus ou moins :



Vous avez sans doute remarqué que, plus vous bougez vite la souris, et plus les points dessinés par votre crayon virtuel s'espacent.

Je vous propose de remplacer la fonction “point” par la fonction “line” qui nous permet de tracer des segments de droite, d'un point X,Y à un autre point X,Y.

```
function setup() {
    createCanvas(600, 400);
}

function draw() {
    if (mouseIsPressed) {
        line(pmouseX, pmouseY, mouseX, mouseY);
    }
}
```

En maintenant le bouton de la souris enfoncé, on peut maintenant écrire ceci :



A quoi correspondent les variables “pmouseX” et “pmouseY” que j’ai utilisées comme premiers paramètres de la fonction “line” ?

Pour rappel, nous savons que la fonction “draw” est lancée en boucle par P5. A chaque fin d’exécution de la fonction “draw”, P5 conserve les dernières coordonnées X, Y connues de la souris. Ce sont ces coordonnées que l’on retrouve lors de l’appel suivant de la fonction “draw”, dans les variables pmouseX et pmouseY (le “p” correspond au mot anglais “previous”, ou “précédent” en français).

Nous pouvons faire plein de choses amusantes avec les variables mouseX et mouseY associées à la fonction “line”. Par exemple, nous pouvons demander à P5 de tracer des lignes qui partent toutes du centre du canvas, jusqu’aux coordonnées courantes de la souris (au moment où je presse le clic gauche de la souris) :

```
function setup() {
    createCanvas(600, 400);
}

function draw() {
    if (mouseIsPressed) {
        line(width/2, height/2, mouseX, mouseY);
    }
}
```

Nous avons vu que P5 met à notre disposition la variable “mouselsPressed”. Il s’agit d’un type de variable particulier, le type “booléen” (en anglais : “Boolean”). Ce type de variable peut contenir les valeurs “true” (vrai) et “false” (faux). Donc si “mouselsPressed” est à “true”, cela signifie que le bouton de la souris est pressé, et que nous pouvons commencer à dessiner.

La structure “if” qui est associée à la variable “mouselsPressed” est une structure conditionnelle...appelée plus communément un “test”.

Donc le code suivant :

```
if (mouseIsPressed) {  
    line(width/2, height/2, mouseX, mouseY);  
}
```

.... peut se traduire d'un point de vue algorithmique par :

Si le bouton de la souris est pressé Alors
 Je trace une ligne partant du centre du canvas et allant au point (mouseX, mouseY)
}

Ah oui, j'ai oublié de préciser un truc : le test suivant...

```
if (mouseIsPressed) {  
    line(width/2, height/2, mouseX, mouseY);  
}
```

... est strictement équivalent à :

```
if (mouseIsPressed == true) {  
    line(width/2, height/2, mouseX, mouseY);  
}
```

Par contre il n'est pas du tout équivalent à ceci :

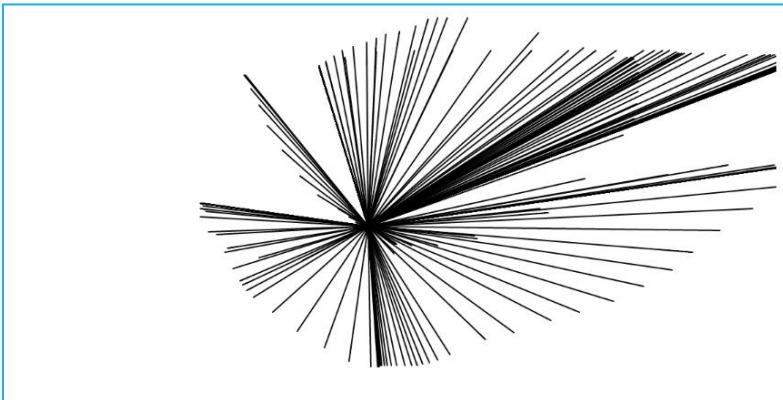
```
if (mouseIsPressed = true) {  
    line(width/2, height/2, mouseX, mouseY);  
}
```

... car dans le test ci-dessus, ce que l'on a écrit c'est :

"la variable mouseIsPressed reçoit la valeur true, et puisque l'affectation de cette valeur true s'est bien passée, alors Javascript renvoie true pour informer le test du fait que l'affectation s'est bien passée (chouette ☺ !!). "

Bref, en utilisant un simple égal au lieu d'un double égal, on vient d'écrire une grosse bêtise. C'est le genre d'erreur qui est difficile à détecter, particulièrement après quelques heures de travail, quand on commence à avoir la vue qui fatigue, et le pire c'est que l'on ne s'en rend pas compte forcément tout de suite. Donc, attention, bannissez le simple égal de vos tests, si vous voulez vous éviter quelques migraines ☺.

Exemple de résultat obtenu avec le sketch qui précède :



Jusqu'ici nous ne sommes pas trop préoccupés de couleurs, car si on ne précise rien P5 va définir un fond blanc et un crayon de couleur noire par défaut.

Allez ! On le refait avec un peu de couleur :

```
function setup() {
    createCanvas(600, 400);
    stroke(0, 10, 90, 10); // rouge 0, vert 10, bleu 90, opacité 10
    background(230); // gris très clair
}

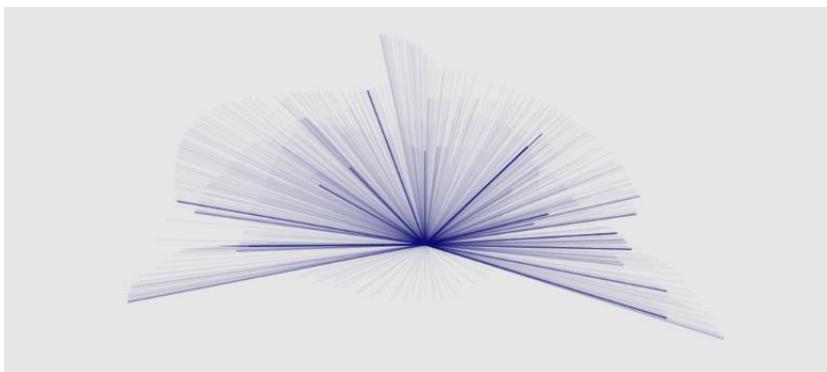
function draw() {
    if (mouseIsPressed) {
        line(width/2, height/2, mouseX, mouseY);
    }
}
```

La fonction “stroke” est une fonction P5 qui nous permet de définir la couleur de notre crayon virtuel.

Les 3 premiers paramètres de la fonction “stroke” correspondent, dans l’ordre, aux niveaux de rouge, de vert et de bleu. Chacun de ces niveaux peut prendre une valeur comprise entre 0 et 255.

Le 4ème paramètre est facultatif (on n’est pas obligé de le définir), il correspond au niveau d’opacité. En le fixant à 10, on obtient un effet de transparence intéressant.

Voici un exemple d'image obtenue avec le sketch que nous venons d'étudier :



J'ai oublié de parler de la fonction "background". C'est elle qui nous permet de définir la couleur du fond, que j'ai paramétrée ici avec un gris très clair. On peut définir 256 niveaux de gris allant de la valeur 0 à la valeur 255, mais on peut aussi définir des couleurs avec la fonction "background", en utilisant 3 ou 4 paramètres, exactement comme on l'a fait avec la fonction "stroke".

Amusez-vous à modifier les valeurs des paramètres transmis aux fonctions "stroke" et "background". Prenez le temps de lire la documentation de ces fonctions sur le site officiel de P5, vous y verrez d'autres manières de définir les couleurs dont je n'ai pas parlé ici:

<http://p5js.org/reference/>

Nous avons vu beaucoup de notions dans ce chapitre, avec quelques exemples de sketches permettant de dessiner des formes très simples. Nous avons abordé brièvement la notion de test (le fameux "if"), abordé les fonctions "stroke" et "background" et la notion de couleur. Nous avons même évoqué les notions d'opacité et de transparence.

2.3 Approfondissement sur la notion de variable

Nous allons approfondir la notion de variable, que nous avons pour l'instant juste entreaperçue.

Si vous êtes un développeur aguerri, vous pouvez allègrement “sauter” ce chapitre ☺.

Si vous êtes débutant en programmation, peut être vous demandez-vous ce qu'est une variable. Une variable, c'est un petit bout de la mémoire de votre ordinateur, qui porte un nom (on l'appelle quelquefois “étiquette”) et contient une valeur.

Si vous cherchez une analogie avec le monde physique, imaginez que la mémoire de l'ordinateur est une gigantesque armoire avec plein de tiroirs. Chaque tiroir porte une étiquette qui précise son contenu, et contient une valeur.

Par exemple, dans mon armoire imaginaire :

- le tiroir qui s'appelle “titi_couleur” contient le mot “jaune”,
- le tiroir qui s'appelle “titi_passe_temps” contient la phrase : “embêter gros minet”,
- le tiroir “gros_minet_passe_temps” contient la phrase : “essayer de bouloter titi”,
- le tiroir qui s'appelle “titi_score” contient la valeur 20000
- le tiroir qui s'appelle “gros_minet_score” contient la valeur 0.

En Javascript, on pourrait écrire :

```
var titi_couleur = 'jaune';
var titi_passe_temps = 'embêter gros minet';
var gros_minet_passe_temps = 'essayer de bouloter titi';
var titi_score = 20000;
var gros_minet_score = 0;
```

Je vous propose de repartir de l'un des sketchs du chapitre précédent :

```
function setup() {
    createCanvas(600, 400);
}
function draw() {
    if (mouseIsPressed) {
        line(width/2, height/2, mouseX, mouseY);
    }
}
```

Nous avons déjà manipulé quelques variables, sans le savoir, puisque “mouseX” et “mouseY” sont des variables fournies par P5. De même, “width” et “height”, qui définissent respectivement la largeur et la hauteur du canvas, sont des variables elles

aussi fournies par P5. Elles correspondent strictement aux valeurs que vous avez transmises à la fonction “CreateCanvas” au tout début de la fonction “setup”.

Vous pourriez vous demander quel est l'intérêt des variables “width” et “height”... bonne question ! Reprenons l'exemple du sketch précédent et remplaçons “width” et “height” par leurs valeurs réelles (je les mets en rouge pour faciliter le repérage dans le code) :

```
function setup() {
    createCanvas(600, 400);
}

function draw() {
    if (mouseIsPressed) {
        line(600/2, 400/2, mouseX, mouseY);
    }
}
```

Vous remarquerez que j'aurais pu écrire “300” au lieu de “600/2” et “200” au lieu de “400/2”, ce qui aurait permis d'éviter 2 divisions pas franchement utiles.

Le sketch que je viens de modifier produira le même résultat que le sketch précédent. Mais imaginez qu'à un moment vous décidiez de modifier les valeurs transmises à la fonction “createCanvas”. Voici quelques exemples que vous pourriez avoir envie d'essayer :

```
createCanvas(1200, 800); // canvas 2 fois plus grand
createCanvas(windowWidth, windowHeight); // canvas de la taille de l'écran
```

Avec notre toute dernière version du sketch, nous nous trouvons dans l'obligation de réajuster les valeurs transmises à la fonction “line”. Dans le cas de ce sketch très simple, la modification est rapide, mais je vous garantis que très vite, vos sketches vont s'étoffer, et que vous allez en avoir vite assez de réajuster des dimensions que vous auriez pu stocker en variables (sans compter les risques d'oubli et d'erreur).

P5 fournit un jeu de variables spécifiques, on en a vu un petit échantillon avec “width”, “height”, “mouseX”, etc... Vous pouvez définir vos propres variables, pour stocker les informations dont vous avez besoin.

En Javascript, pour créer vos propres variables, vous devez utiliser le mot clé “var” devant la variable à déclarer. Par exemple, pour déclarer la variable xcentre, on écrira :

```
var xcentre;
```

On peut aussi profiter de la déclaration de la variable pour lui attribuer une valeur par défaut (par exemple, zéro) :

```
var xcentre = 0;
```

Le sketch ci-dessous présente un petit défaut que j'aimerais bien corriger :

```
function setup() {
    createCanvas(600, 400);
}

function draw() {
    if (mouseIsPressed) {
        line(width/2, height/2, mouseX, mouseY);
    }
}
```

Vous voyez que dans la fonction “draw”, nous effectuons 2 calculs qui sont : “width / 2” et “height / 2”. Cela nous permet de connaître les coordonnées X,Y du centre du canvas.

Sachant que la fonction “draw” est exécutée par P5 environ 60 fois par seconde, et que la taille de l’écran est fixe (600 sur 400 pixels dans notre exemple), vous comprenez bien que l’on répète ces 2 calculs environ 60 fois par seconde : cela fait donc environ 120 calculs inutiles par seconde, et si vous laissez votre sketch “tourner” pendant 1 heure, je vous laisse faire le calcul du nombre d’opérations inutiles réalisées par le moteur Javascript de votre navigateur préféré. On pourrait essayer de le soulager un peu ce pauvre moteur JS non ?

Sachant que la fonction “setup” n'est exécutée qu'une seule fois, cela semble être le bon endroit pour précalculer les coordonnées X,Y du centre du canvas, et par la même occasion stocker ces informations dans nos propres variables... Essayons et voyons ce qui se passe :

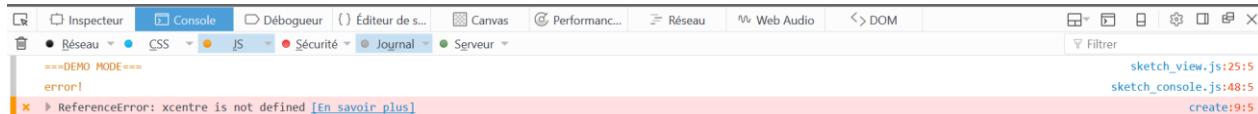
```
function setup() {
    createCanvas(600, 400);
    var xcentre = width/2;
    var ycentre = height/2;
}

function draw() {
    if (mouseIsPressed) {
        line(xcentre, ycentre, mouseX, mouseY);
    }
}
```

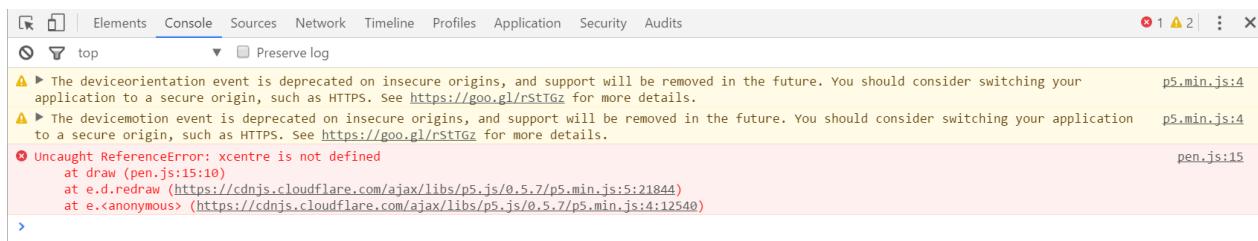
Je vous invite à tester cette nouvelle version de sketch, dans Codepen ou dans Openprocessing, selon votre préférence... Rien ne se passe quand vous cliquez sur l'écran? Hmm, cela signifie que nous avons très certainement commis une erreur au niveau du code Javascript, erreur qui a pour effet de bloquer l'exécution du sketch. Il faut que nous allions voir ce qui se passe sous le capot du navigateur.

Pour ausculter notre navigateur, nous devons accéder aux outils de développement que le navigateur met à notre disposition. Sur de nombreux navigateurs (Google Chrome, Firefox, Internet Explorer...) ces outils de développement sont accessibles en pressant la touche F12. Dans ces outils de développement, la partie qui va nous intéresser le plus, c'est la console, aussi je vous invite à cliquer sur l'onglet “console” :

Voici ce que vous allez trouver sur Firefox :



Sur Google Chrome, l'affichage de la console indique ceci (c'est surtout le message en rouge qui nous intéresse) :



On voit très clairement que le problème se situe sur la variable xcentre. Les navigateurs nous indiquent en effet que cette variable n'est pas définie. Etant donné que nous la

définissons dans la fonction “setup” et que nous l'utilisons dans la fonction “draw”, c'est de toute évidence l'utilisation de cette variable dans “draw” qui pose problème.

Pour comprendre ce qui se passe, il est temps d'introduire la notion de “portée”. La portée d'une variable, c'est son périmètre d'action. Une variable qui est déclarée à l'intérieur d'une fonction n'est accessible qu'à l'intérieur de cette fonction. On dit que la portée de la variable est “locale” à la fonction dans laquelle elle est déclarée. Je n'entre pas ici dans le détail des fonctions contenant d'autres fonctions, je préfère demeurer sur un cas simple dans un premier temps.

La variable `xcentre` étant déclarée dans la fonction “setup”, elle n'existe pas pour la fonction “draw”. Si l'on souhaite que la variable `xcentre` puisse être utilisée à la fois dans la fonction “setup” et dans la fonction “draw”, alors nous devons la déclarer en dehors de ces deux fonctions, elle sera alors considérée comme une variable “globale” par les deux fonctions qui pourront donc en disposer.

Je vous propose d'appliquer à notre sketch précédent la modification suivante :

```
var xcentre, ycentre; // déclaration des 2 variables globales

function setup() {
    createCanvas(600, 400);
    xcentre = width/2;
    ycentre = height/2;
}

function draw() {
    if (mouseIsPressed) {
        line(xcentre, ycentre, mouseX, mouseY);
    }
}
```

Testez cette nouvelle version de sketch, il devrait de nouveau fonctionner normalement.

Point important : vous remarquerez que j'ai retiré le mot clé “var” devant les variables “`xcentre`” et “`ycentre`”, à l'intérieur de la fonction “`setup`”. Si j'avais oublié de les enlever, alors les variables “`xcentre`” et “`ycentre`” déclarées à l'intérieur de la fonction “`setup`” auraient été prioritaires sur celles de même nom déclarées en dehors. Je vous invite à faire volontairement l'erreur, et à regarder ce qui se passe... vous verrez, c'est très instructif ☺.

Il y a encore beaucoup de choses à dire sur les variables, nous n'avons fait qu'effleurer le sujet, et nous y reviendrons.

Avant de poursuivre notre exploration de P5, retournons un instant dans la console du navigateur (Firefox ou Chrome, selon votre préférence). Cette console est vraiment très pratique. Vous pouvez par exemple l'utiliser comme calculette :

```
> 5+3*2/20
< 5.3
> (5+3)*2/20
< 0.8
> |
```

... effectuer des calculs scientifiques :

```
> 2 * Math.PI + Math.sin(1) * 5
< 10.490540231219068
>
```

... lui demander la date et l'heure courante :

```
> new Date
< Tue Mar 14 2017 23:18:01 GMT+0100 (Paris, Madrid)
```

La console Javascript est l'amie des développeurs web et en particulier des développeurs Javascript, je l'utilise tous les jours dans le cadre de mon travail , c'est vraiment un super outil.

A chaque fois qu'un sketch ne produit pas les effets attendus, et si l'origine du problème ne vous apparaît pas clairement, ayez le réflexe de lancer la console de votre navigateur.

J'avais l'air de plaisanter en parlant de "soulager le moteur JS" des opérations inutiles, mais en réalité c'est un sujet très sérieux. Réduire les operations inutiles a 2 effets importants :

- cela permet d'accélérer la vitesse d'exécution de votre sketch, donc d'améliorer la fluidité des animations (sujet que nous aborderons bientôt),
- cela permet de réduire l'activité du processeur, et en particulier cela lui évite de surchauffer. S'il surchauffe, il consomme plus d'électricité, et votre ordinateur va devoir le refroidir, donc consommer lui aussi plus d'électricité. Donc, réduire l'activité du processeur aux seules opérations utiles, c'est bon pour la planète.

Nous avons vu beaucoup de choses dans ce chapitre. Nous avons approfondi la notion de variable, évoqué la notion de calcul et le stockage du résultat de ces calculs dans des variables. Nous avons brièvement abordé la notion d'optimisation – et même d'écologie - en essayant d'éliminer des calculs inutiles. Nous avons vu quelques unes des nombreuses possibilités de la console Javascript... N'hésitez pas à faire une pause, et à relire ce chapitre si certaines notions vous semblent encore confuses. Ne vous inquiétez pas si tout ne vous semble pas encore clair, vous connaissez sans doute l'expression "Paris ne s'est pas faite en un jour" ☺.

2.4 La fonction Keypressed est votre amie

Avant de poursuivre plus avant l'exploration de P5, je vous propose de nous arrêter quelques instants sur la fonction “keyPressed”.

Cette fonction est une fonction que P5 propose de gérer pour nous, si l'on décide de l'utiliser. Elle est en effet facultative, libre à nous de l'utiliser ou pas.

Pour ma part, j'ai pris l'habitude de l'intégrer à la plupart de mes sketchs, quelquefois avec de légères variantes, mais elle ressemble le plus souvent à ceci :

```
function keyPressed() {  
    if (key == 'x' || key == 'X') {  
        noLoop();  
    } else {  
        if (key == 's' || key == 'S') {  
            saveCanvas("img-" + frameCount + ".png");  
        }  
    }  
}
```

Dans la fonction keyPressed ci-dessus, je teste si l'utilisateur a pressé la touche « s » (en majuscule ou en minuscule). Si c'est le cas, je fais appel à la fonction « noLoop ». Il s'agit d'une fonction fournie par P5 qui a pour effet de stopper l'exécution en boucle de la fonction « draw ». C'est très pratique si vous pensez que votre sketch contient une anomalie, et que vous souhaitez le stopper avant d'analyser son code.

Autre fonctionnalité que j'aime bien implémenter dans la fonction « keyPressed », c'est la possibilité de sauvegarder l'image en cours à tout moment. J'associe cette action aux touches « s » majuscule et minuscule. Pour le nom du fichier sauvegardé, j'ai concaténé quelques bouts de chaînes de caractère (« img », « .png ») avec la variable « frameCount ». Il s'agit d'un compteur que P5 gère pour nous et qui contient le numéro de « frame » (d'image) générée par P5 pendant l'exécution du sketch. C'est une manière de faire... il y en a d'autres, vous pourrez le modifier plus tard si vous le jugez utile.

On peut revenir un instant sur la structure conditionnelle précédente, car elle contient des éléments que nous n'avons pas encore vu :

```
if (key == 'x' || key == 'X') {  
    noLoop();  
} else {  
    if (key == 's' || key == 'S') {  
        saveCanvas("img-" + frameCount + ".png");  
    }  
}
```

Vous voyez que le « ou » s'écrit en JS en utilisant la double barre verticale (en anglais : « double pipe », prononcé « deubeul païpe »).

Pour écrire un « et » en JS, on utilise le symbole suivant : `&&`

Vous voyez aussi dans cet exemple nous avons 2 tests liés l'un à l'autre, le second ne s'exécutant que dans le cas où la condition du premier test n'est pas remplie. C'est le « else » (sinon) qui nous permet d'écrire cela. C'est une structure à retenir, vous la retrouverez souvent.

2.5 Premier effet d'animation

Les quelques effets d'animation que nous avons implémenté jusqu'ici étaient tous associés aux mouvements de la souris. Mais il y a de nombreux cas où l'on souhaite programmer des effets d'animation qui sont générés automatiquement, sans action de la part de l'utilisateur.

Dans le sketch suivant, nous générerons une multitude de lignes, qui se superposent jusqu'à "recouvrir" l'ensemble du canvas :

```
function setup() {
    createCanvas(300, 200);
}

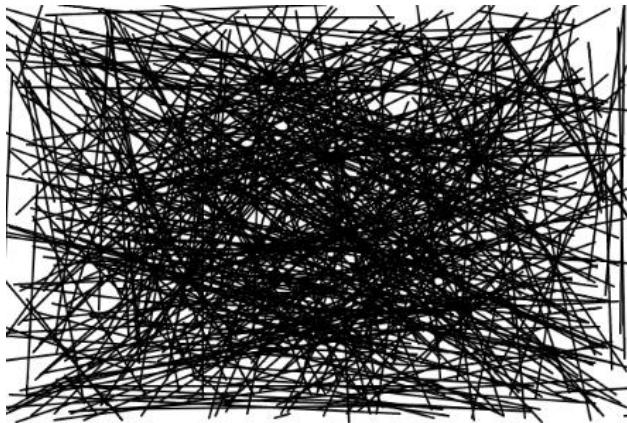
function draw() {
    background(255, 10);
    var x1 = random(0, width-1);
    var y1 = random(0, height-1);
    var x2 = random(0, width-1);
    var y2 = random(0, height-1);
    line(x1,y1,x2,y2);
}
```

Vous voyez ici que nous alimentons 4 variables correspondant à 2 jeux de coordonnées XY. Les variables x1 et y1 correspondent aux coordonnées d'une extrémité de la ligne à dessiner, les variables x2 et y2 correspondent aux coordonnées de l'autre extrémité.

La fonction "random" est une fonction P5 qui nous renvoie une valeur aléatoire (prise au hasard) entre les limites que nous lui fixons, par exemple "0" et "width-1".

La variable "width" est une variable P5 définissant la largeur de l'écran en pixels. Pourquoi avoir demandé "width-1" et pas tout simplement "width" ? Il faut savoir que le premier point de chaque ligne démarre en position zéro. Donc pour un canvas de 255 pixels de large, nous avons bien 255 pixels, numérotés de zéro à 254. Ainsi, le point en position horizontale 255 se trouve en dehors du canvas... à un pixel près. Avec certains systèmes graphiques plus anciens, demander à tracer un point en dehors de la zone d'affichage entraînait une erreur. P5 est très tolérant et nous pouvons sans souci transmettre aux fonctions de dessin (comme "line" et "point") des coordonnées se situant en dehors du canvas, P5 ne les dessinera pas, un point c'est tout.

Au bout de quelques secondes d'exécution du sketch précédent, nous obtenons une image qui ressemble plus ou moins à ceci (elle sera différente à chaque nouvelle exécution, du fait de l'utilisation de la fonction "random") :



Nous pouvons faire appel à la fonction P5 “background” en la plaçant au début de la fonction “draw”, comme dans l'exemple ci-dessous. Cette fonction “background” a pour effet de réinitialiser le canvas en appliquant la couleur qui lui est transmise en paramètre. La valeur “255” qui est transmise dans l'exemple ci-dessous correspond à la couleur blanche. L'écran est donc remis à blanc à chaque nouvel appel de la fonction “draw”.

```
function setup() {
    createCanvas(300, 200);
}

function draw() {
    background(255);
    var x1 = random(0, width-1);
    var y1 = random(0, height-1);
    var x2 = random(0, width-1);
    var y2 = random(0, height-1);
    line(x1,y1,x2,y2);
}
```

On constate que les lignes tracées aléatoirement sont effacées très rapidement. L'effet est peu intéressant d'un point de vue visuel, il serait plus intéressant que l'effacement des segments soit progressif. Nous allons obtenir cet effet en jouant sur le second paramètre de la fonction `background`, qui permet de définir l'opacité. Dans l'exemple qui précède, modifiez la ligne contenant la fonction “background” de la façon suivante :

```
background(255, 20);
```

Relancez l'exécution de votre sketch et observez ce qui se passe. Puis modifiez la valeur du second paramètre, en testant différentes valeurs entre 1 et 120.

Cette technique permet d'obtenir des effets graphiques intéressants avec très peu de moyens, je l'avais découverte dans le livre « Processing » de Casey Reas et Ben Fry.

Nous ne pourrons pas recourir à cet effet à tous les coups, dans certains cas nous devrons recourir à d'autres techniques que nous aborderons dans quelques instants.

Mais avant de passer au chapitre suivant, je voudrais revenir sur un point. L'exemple suivant est presque identique à l'exemple précédent, à la différence près que la fonction "random" est encapsulée dans une fonction "int" :

```
function setup() {
    createCanvas(300, 200);
}

function draw() {
    background(255);
    var x1 = int(random(0, width-1));
    var y1 = int(random(0, height-1));
    var x2 = int(random(0, width-1));
    var y2 = int(random(0, height-1));
    line(x1,y1,x2,y2);
}
```

Sachant que la fonction "random" renvoie des valeurs contenant des décimales, la fonction "int" a pour effet de "renvoyer" la partie entière des nombres générés par "random". Dans le cas présent, c'est un peu superflu car la fonction "line" fournie par P5 effectue d'office ce petit ménage. Il est néanmoins intéressant de connaître cette forme d'écriture, vous retrouverez souvent cette possibilité d'encapsuler l'appel de fonctions à l'intérieur d'autres fonctions.

2.6 Créer sa propre fonction « line »

Nous avons vu que nous pouvons faire appel à des fonctions P5 telles que “point” et “line”. La fonction “line” est vraiment très pratique, mais dans certains cas on peut avoir besoin de créer une fonction “line” maison, afin de réaliser des effets que ne permet pas de produire la fonction “line” standard. Par exemple, on peut souhaiter tracer une ligne en pointillés, ou une ligne avec une alternance de couleurs pour les points pairs et impairs de la ligne.

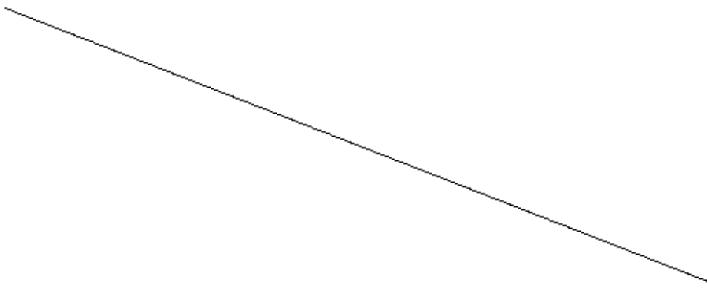
Mais au fait, comment fait-on pour dessiner des segments de droite dans un canvas ? Cela semble relativement simple si ces segments sont strictement horizontaux ou verticaux, mais s’ils sont obliques ? Le sketch qui suit propose un exemple d’algorithme répondant à ce besoin. Je vous propose de le tester, ensuite nous détaillerons son fonctionnement.

```
var sx = 600, sy = 500;
var backcolor = "white", drawcolor = "black";
var x1=10, y1=10, x2=500, y2=200; // valeurs de test
var pas_increment = 1;

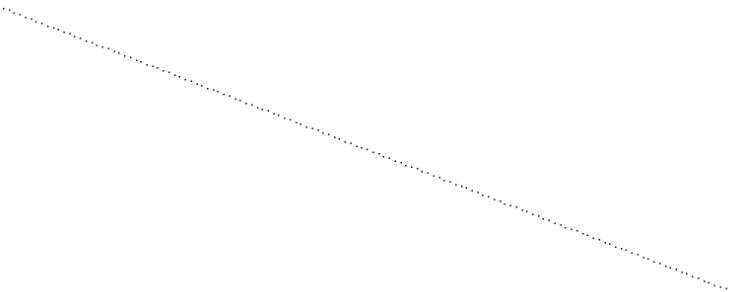
function setup() {
    createCanvas(sx, sy);
    background(backcolor);
    stroke(drawcolor);
}

function draw() {
    var dx = x2 - x1;
    var dy = y2 - y1;
    var d = sqrt(dx * dx + dy * dy);
    var sx = dx / d;
    var sy = dy / d;
    for (var i = 0 ; i <= d ; i += pas_increment) {
        point ( x1+sx*i , y1+sy*i );
    }
    noLoop(); // pas de boucle
}
```

C'est prêt ? Alors testons-le :



C'est cool, et si on modifiait la valeur de la variable “pas_increment”, par exemple en la fixant à 4 :



Dans ce sketch, nous avons des variables globales (comme “sx”, “sy”, ..) et locales (comme “dx”, “dy”, ..). Les variables globales sont celles qui se trouvent en dehors des fonctions “setup” et “draw”. Les variables locales sont celles qui se situent à l’intérieur de fonctions.

Nous avons également des additions, des soustractions, des divisions et des multiplications. Si vous n’êtes pas très familier avec cette forme d’écriture, retenez simplement que :

```
var dx = x2 - x1;
```

... se lit de la façon suivante :

“la variable dx reçoit le résultat du calcul suivant : soustraction du contenu de la variable x1 au contenu de la variable x2”

Eh oui, le égal se lit ici “reçoit” et non pas “égal”. Comme ce symbole est trompeur, il vaut mieux se le représenter mentalement comme une flèche allant de la droite vers la gauche :

```
var dx <- x2 - x1;
```

Autre nouveauté, nous avons fait appel à la fonction “racine carrée”, qui s’écrit en JS “Math.sqrt()”, mais P5 nous fournit un raccourci avec la fonction “sqrt()” :

```
var d = sqrt(dx * dx + dy * dy);
```

Enfin, nous avons écrit une boucle “for” qui est une structure que l’on retrouve dans la plupart des langages de programmation :

```
for (var i = 0 ; i <= d ; i += pas_increment) {  
    point ( x1+sx*i , y1+sy*i );  
}
```

Une boucle comme celle-ci se lit de la façon suivante :

Pour toute valeur de “i” comprise entre “zéro inclus” et “d inclus”, avec i incrémentée de la variable “pas_increment” à chaque itération

Tracer un point aux coordonnées indiquée entre parenthèses

Fin de Pour

Peut être à ce stade souhaitez-vous mieux comprendre ce qui se passe à l’intérieur de la boucle. Je vous propose dans ce cas d’essayer la fonction JS “console.log”. Il s’agit d’une fonction qui est embarquée dans tous les navigateurs. Pour ce faire, ajoutez une ligne à l’intérieur de votre boucle “for” :

```
for (var i = 0 ; i <= d ; i += pas_increment) {  
    point ( x1 + sx * i , y1 + sy * i );  
    console.log ( i + ' <- ' + (x1 + sx * i) + ' , ' + (y1 + sy * i));  
}
```

La fonction console.log reçoit une chaîne de caractères comme paramètre d’entrée. Dans cette chaîne de caractères, nous avons concaténé plusieurs informations, que nous avons envoyées vers la console du navigateur. La concaténation se fait en JS avec le symbole “+”. La concaténation, si vous ne savez pas ce que c’est, c’est l’action de mettre bout à bout plusieurs informations (numériques ou caractères) pour obtenir une chaîne de caractères en sortie.

Activez la console du navigateur (via la touche F12), lancez l’exécution de votre sketch... voici un extrait de ce que vous allez y trouver :

```
"0 <- 10 , 10"  
"4 <- 13.729445235800577 , 11.44611141796349"  
"8 <- 17.458890471601155 , 12.89222283592698"  
"12 <- 21.188335707401734 , 14.338334253890467"  
"16 <- 24.91778094320231 , 15.784445671853957"  
"20 <- 28.647226179002885 , 17.230557089817445"  
..."  
"516 <- 491.0984354182745 , 196.54837291729012"  
"520 <- 494.82788065407505 , 197.99448433525362"  
"524 <- 498.55732588987564 , 199.4405957532171"
```

On voit que l'on part bien du point de coordonnées 10,10, pour arriver au point de coordonnées 500, 200... enfin presque, car avec les problèmes d'arrondi dans les calculs, le résultat n'est pas parfait.

Si l'on fixe la variable "pas_increment" à 1, alors le dernier point généré se rapproche davantage de la limite maximale fixée :

```
525 <- 499.4896871988258 , 199.80212360770798
```

Intéressant non ? Vous voyez qu'avec la fonction "console.log", vous pouvez faire "dire" beaucoup de choses au navigateur, il n'a presque plus de secrets pour vous ☺.

2.7 Un peu de communication avec l'utilisateur

Partant du sketch précédent, j'aimerais que l'utilisateur puisse saisir lui-même les coordonnées X,Y des extrémités de la ligne à dessiner.

Il existe différentes manières d'interagir avec un utilisateur, et de l'amener à nous fournir des informations. Le plus souvent cela passe par la gestion d'événements tels que des clics sur des boutons, ou des saisies d'informations dans des champs de formulaire.

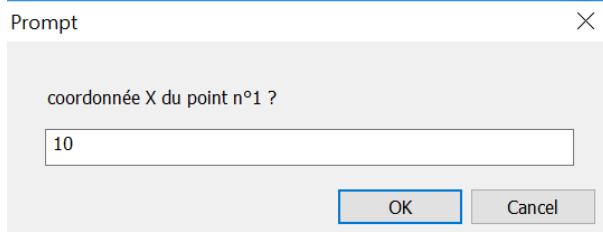
Mais je trouve que c'est un peu compliqué d'aborder ces techniques dans le cadre d'une initiation à la programmation, car cela implique d'acquérir certaines notions de HTML, et de maîtriser certaines techniques JS de niveau avancé.

Je vous propose plutôt d'utiliser une bonne vieille technique de saisie, la fonction JS "prompt". Ce n'est pas élégant, mais pour un prototype d'application, c'est rapide et efficace.

Par exemple, l'instruction suivante :

```
var x1 = prompt('coordonnée X du point n°1 ? ', '10');
```

... fait apparaître la fenêtre suivante :



Vous pouvez modifier si vous le souhaitez la valeur par défaut qui est proposée ici ("10"), ensuite cliquez sur "OK", le résultat de votre saisie se trouve automatiquement stocké dans la variable "x1".

Attention, si vous cliquez sur le bouton "Cancel" (annuler), vous allez récupérer une valeur bizarre, la valeur "null".

On pourrait se dire que ce n'est pas un problème, et encapsuler notre fonction "prompt" à l'intérieur d'une fonction "parseInt", comme ceci :

```
var x1 = parseInt(prompt('coordonnée X du point n°1 ? ', '10'));
```

La fonction "parseInt" est une fonction JS qui recherche la première valeur numérique stockée dans une chaîne de caractères. Elle fonctionne plutôt bien dans de nombreux

cas, mais elle ne sait pas quoi faire d'une valeur "null". Et quand "parseInt" ne sait pas quoi faire, elle nous l'indique en nous renvoyant la valeur "NaN" (pour "Not a Number").

Si dans le formulaire "prompt" vous saisissez "20" ou "20abc", la fonction "parseInt" va être en mesure de vous renvoyer la valeur 20. En revanche, si vous saisissez "toto" ou "a20", la fonction "parseInt" va vous renvoyer cette fameuse valeur "NaN".

Il existe une fonction "Number.isNaN", qui va nous permettre de vérifier la validité de la saisie de l'utilisateur. Grâce à cette fonction, nous pouvons écrire un petit bout de code qui va tester la validité de la saisie de l'utilisateur, et "tourner en boucle" jusqu'à ce que la saisie soit correcte :

```
var x1 = NaN;  
while (Number.isNaN(x1)) {  
    x1 = parseInt(prompt('coordonnée X du point n°1 ? ', '10'));  
}
```

Ainsi, si l'utilisateur clique sur le bouton "cancel", ou s'il saisit n'importe quoi, il restera bloqué dans le formulaire de saisie. Je vous avais prévenu, ce n'est pas élégant, mais c'est simple et efficace 😊.

Jusqu'ici nous avions vu la boucle de type "for", mais pas la boucle de type "while". Dans notre cas, on pourrait traduire le code ci-dessus de cette manière :

```
x1 est initialisée avec la valeur "Not a Number"  
Tant_QUE x1 est égale à "Not a Number"  
    Afficher le formulaire de saisie, récupérer la saisie  
    de l'utilisateur, convertir cette saisie en numérique,  
    et stocker le résultat dans x1  
Refaire
```

Attention, il existe en JS une fonction "isNaN" (ou "window.NaN" qui est synonyme). Mais cette fonction n'est pas fiable, comme l'avait souligné Kyle Simpson dans cet excellent livre :

"*You don't know JS : Types and Grammars*", édité chez O'Reilly

Il est donc préférable d'utiliser la fonction "Number.isNaN", comme je l'ai présenté dans l'exemple précédent. Si vous souhaitez approfondir ce sujet des variables numériques "NaN", je vous invite à petit détour en annexe (cf. chapitre 5.1).

Si l'on en revient à notre sketch de génération de lignes, et que nous y ajoutons la possibilité de saisir les coordonnées cartésiennes de notre ligne, nous pouvons écrire quelque chose de ce genre :

```
var sx = 600;
var sy = 500;
var backcolor = "white";
var drawcolor = "black";
var x2, x1, y2, y1;
var pas_increment = 1;

function setup() {
    createCanvas(sx, sy);
    background(backcolor);
    stroke(drawcolor);
    x1 = NaN;
    while (Number.isNaN(x1)) {
        x1 = parseInt(prompt('coordonnée X du point n°1 ? ', '10'));
    }
    y1 = NaN;
    while (Number.isNaN(y1)) {
        y1 = parseInt(prompt('coordonnée Y du point n°1 ? ', '10'));
    }
    x2 = NaN;
    while (Number.isNaN(x2)) {
        x2 = parseInt(prompt('coordonnée X du point n°2 ? ', '500'));
    }
    y2 = NaN;
    while (Number.isNaN(y2)) {
        y2 = parseInt(prompt('coordonnée Y du point n°2 ? ', '200'));
    }
}

function draw() {
    var dx = x2 - x1;
    var dy = y2 - y1;
    var d = sqrt(dx * dx + dy * dy);
    var sx = dx / d;
    var sy = dy / d;
    for (var i = 0 ; i <= d ; i += pas_increment) {
        point ( x1 +sx * i , y1 + sy * i );
    }
    noLoop(); // pas de boucle
}
```

Je pourrais écrire plusieurs pages sur la manière d'optimiser ce sketch, mais nous avons vu beaucoup de choses dans ce chapitre, et je pense que c'est bien que nous en restions là pour l'instant.

Si l'on résume un peu, nous avons vu comment :

- gérer un formulaire de saisie avec la fonction JS “prompt”
- contrôler la validité d'une saisie, avec notamment les fonctions JS “Number.isNaN” et “parseInt”
- écrire une boucle de type “while”

Vous avez le droit de faire une pause, vous l'avez bien méritée ☺.

2.8 Des algorithmes qui dessinent sous nos yeux

On dit qu'un bon dessin vaut mieux qu'un long discours, mais pour faciliter la compréhension d'un concept, il me semble que rien ne vaut un bon dessin animé. Eh puis... voir une forme se dessiner progressivement sur l'écran, c'est vraiment plus fun.

Jusqu'ici nous avons vu quelques algorithmes permettant de dessiner des formes simples, mais il manquait à tout cela cet élément essentiel qu'est le mouvement. Ou pour être plus précis, nous avons bien vu des formes bouger sur l'écran, mais il s'agissait de formes « finies », que nous faisions bouger par exemple avec la position de la souris.

J'aimerais maintenant vous montrer comment construire des formes qui se dessinent progressivement à l'écran, et pour illustrer mon propos je vous propose d'apprendre à construire des cercles. Quand vous maîtriserez le principe, vous pourrez l'appliquer à n'importe quelle forme.

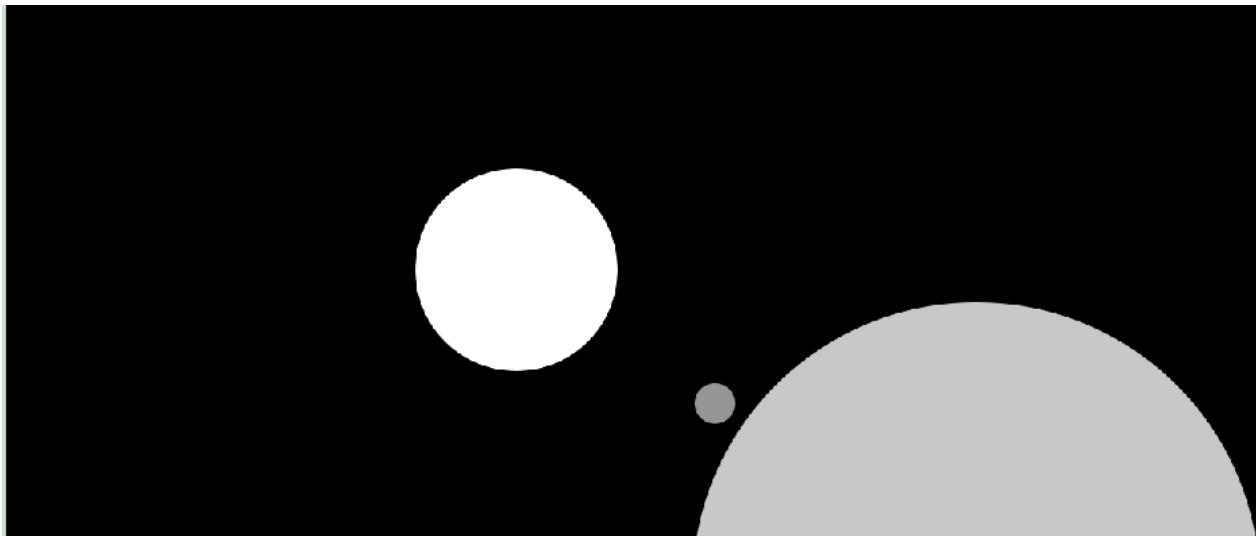
2.8.1 Les primitives proposées par P5

Pour dessiner des formes circulaire, P5 met à notre disposition deux fonctions, « `ellipse` » et « `arc` » :

```
ellipse(x, y, w, [h])
arc(x, y, w, h, angleStart, angleStop, [mode])
```

On peut utiliser ces fonctions pour dessiner un paysage spatial minimaliste comme dans l'exemple suivant :

```
function setup() {
  createCanvas(620, 200); // équivalent Processing : size(620, 300);
  background(0);
  noStroke();
  // lune blanche
  fill(255);
  ellipse(252, 144, 100, 100);
  // moitié supérieure d'une grosse planète gris clair
  fill(200);
  arc(479, 300, 280, 280, PI, TWO_PI);
  // petite lune gris foncé
  fill(150);
  arc(width/2+40, height/2+60, 20, 20, 0, TWO_PI, CHORD);
}
```



Trois petites planètes... sympathique mais un peu statique, pas vrai ?

Voici un autre exemple de sketch beaucoup plus animé. Je vous recommande de l'essayer dans Codepen ou Openprocessing, l'effet est sympa :

```
function setup() {
  createCanvas(620, 300);
  background(0);
  noStroke();
  frameRate(20);
}

// génération de 100 cercles placés aléatoirement
function draw() {
  var posx = int(random(0, width));
  var posy = int(random(0, height));
  var ray = int(random(0, 100));
  fill(random(1, 255));
  ellipse(posx, posy, ray, ray);
  if (frameCount > 100) noLoop();
}
```



Si vous relancez ce sketch plusieurs fois, vous n'obtiendrez jamais les mêmes images, car on a fait ici un usage intensif de la fonction « random » qui nous renvoie des valeurs aléatoires à chaque appel (en fonction des limites minimum et maximum qui lui sont fournies).

Vous noterez la présence d'un test à la fin de la fonction « draw ». La présence de ce test garantit que la fonction « draw » ne sera pas lancée plus de 100 fois. A partir du 101^{ème} appel de « draw », la fonction « noLoop » est appelée, ce qui pour effet de stopper la boucle d'exécution de la fonction « draw ».

2.8.2 Tracé de cercle – algo n° 1 (avec racine carrée)

Il existe au moins deux méthodes algorithmiques permettant de dessiner des cercles.

Voici un premier algorithme, très simple, qui utilise la fonction « sqrt » (racine carrée), comme seul élément mathématique un peu complexe. On a ici une mise en application du fameux « Théorème de Pythagore » :

```
var sx = 500;
var sy = 500;
var xc, yc;
var rayon = 200;
var pas = 1;
function setup() {
    createCanvas(sx, sy);
    xc = sx / 2;
    yc = sy / 2;
}
function draw() {
    var px = -rayon;
    var py = 0;
    for (var x = -rayon ; x <= rayon ; x += pas) {
        var y = sqrt(rayon * rayon - x * x);
        line(xc + px, yc + py, x+xc, yc+y);
        line(xc + px, yc - py, x+xc, yc-y);
        px = x;
        py = y;
        console.log(px + ' ; ' + py);
    }
    noLoop();
}
```

Je rappelle que la fonction « sqrt », que nous avions déjà vu dans un sketch précédent, est un raccourci fourni par P5, vers la fonction « Math.sqrt ». Vous pouvez utiliser indifféremment l'une ou l'autre dans vos sketchs P5.

La fonction « noLoop » placée à la fin de la fonction « draw » empêche P5 de relancer la fonction « draw ». C'est volontaire, car en l'état notre algorithme ne nous permet pas de voir notre cercle se dessiner progressivement. Dans un cas comme celui-ci, stopper la boucle d'exécution de la fonction « draw » ne changera rien d'un point de vue visuel, mais cela soulagera l'interpréteur JS qui pourra se consacrer à d'autres tâches.

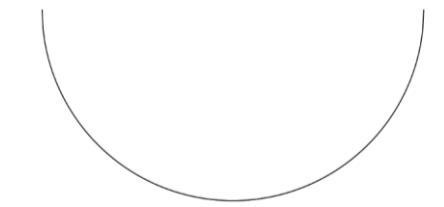
Vous l'aurez sans doute deviné, les lignes stratégiques pour le bon fonctionnement de notre sketch sont les lignes suivantes (qui se trouvent dans la boucle « for ») :

```
var y = sqrt(rayon * rayon - x * x);
line(xc + px, yc + py, x+xc, yc+y);
line(xc + px, yc - py, x+xc, yc-y);
px = x;
py = y;
```

Amusez-vous à mettre la 3^{ème} ligne ci-dessous en commentaire et relancez l'exécution de votre sketch :

```
var y = sqrt(rayon * rayon - x * x);
line(xc + px, yc + py, x+xc, yc+y);
// line(xc + px, yc - py, x+xc, yc-y);
px = x;
py = y;
```

On n'obtient plus que l'affichage du demi-cercle inférieur :

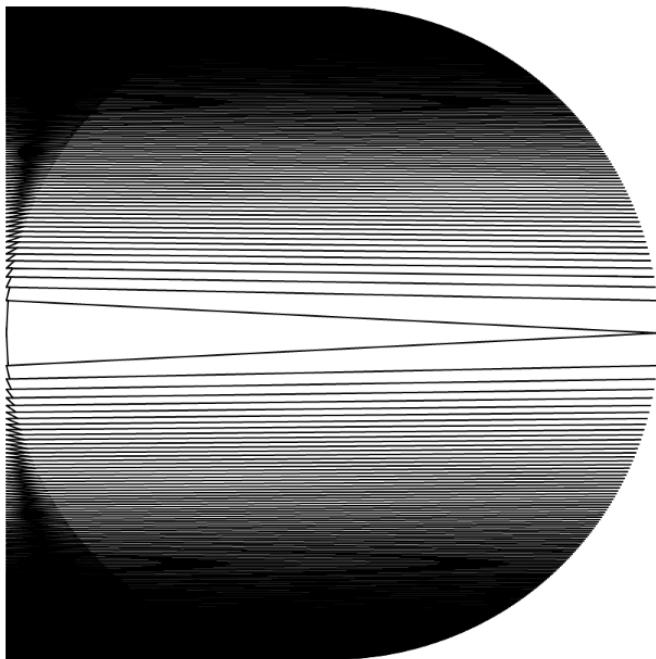


Vous pouvez mettre en commentaire la 2^{ème} ligne, décommenter la 3^{ème}, vous obtiendrez le demi-cercle supérieur.

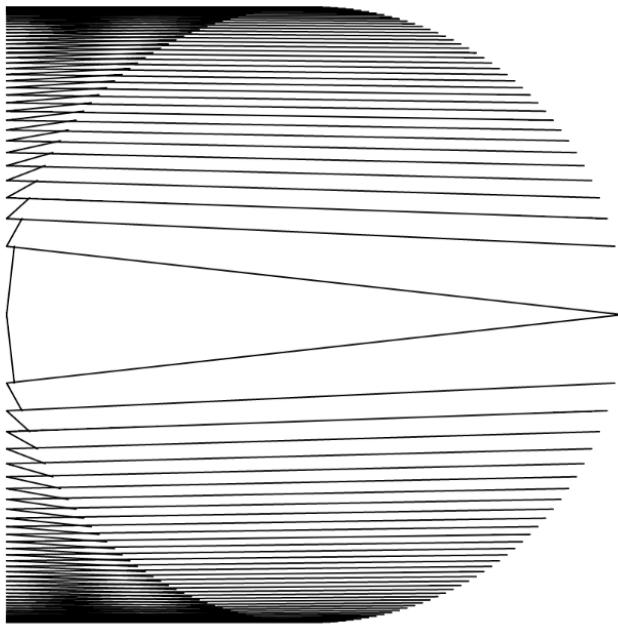
La fonction « console.log » utilisée dans la boucle est facultative. Elle vous permet d'afficher dans la console du navigateur le contenu des variables « px » et « py ». Ces deux variables sont importantes, car elles assurent la continuité des positions entre chaque itération de la boucle « for ». Pour comprendre leur importance, amusons-nous à mettre en commentaire la 4^{ème} ligne de notre portion de code :

```
var y = sqrt(rayon * rayon - x * x);
line(xc + px, yc + py, x+xc, yc+y);
line(xc + px, yc - py, x+xc, yc-y);
//px = x;
py = y;
```

Résultat obtenu ci-dessous... surprenant non ?



Si vous augmentez la valeur de la variable « pas » qui se trouve au tout début du sketch, par exemple en la fixant à 5, vous obtenez un effet plus agréable à l'oeil :

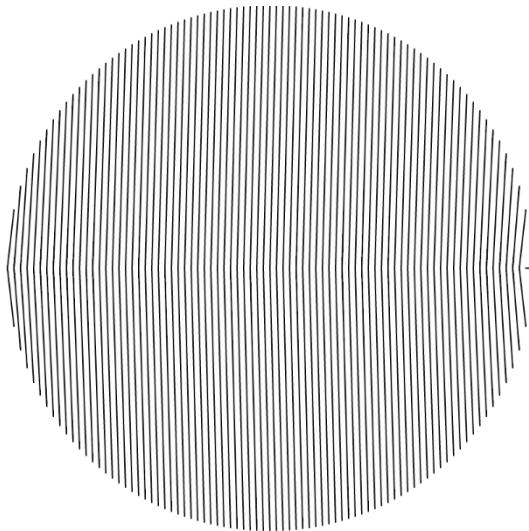


Vous voyez qu'une erreur glissée subrepticement dans le code peut aboutir à des effets surprenants, et quelquefois plus intéressants que le résultat visé initialement.

Si vous inversez les lignes commentées et décommentées :

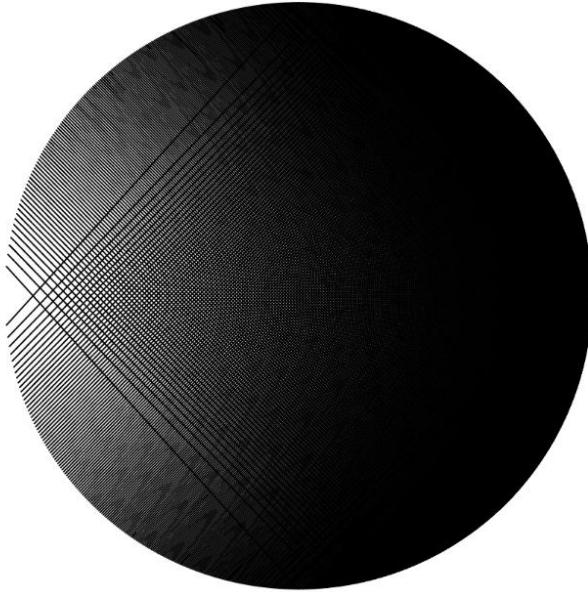
```
//px = x;  
py = y;
```

... alors vous obtenez ceci :



Et si accidentellement vous inversiez les affectations des 2 lignes suivantes, cela donnerait quoi ?

```
px = y; // y placé dans px au lieu de py  
py = x; // x placé dans py au lieu de px
```



Tiens, un conseil, gardez le sketch en l'état et augmentez la valeur de la variable « pas » (par exemple fixez la à 4). Relancez le sketch, vous verrez, l'effet est encore plus intéressant.

Vous venez de mettre un premier pas dans le monde des bugs 😊, mais vous voyez que ce n'est pas toujours aussi désagréable qu'on le dit.

Un petit conseil : si suite à un bug vous obtenez un effet graphique intéressant, notez scrupuleusement quelque part ce qui est à l'origine du bug. Ainsi, si vous souhaitez reproduire par la suite cet effet accidentel qui vous a tant plu, vous saurez comment vous y prendre. Votre serviteur a quelquefois regretté de ne pas avoir appliqué ce principe pour lui-même 😊.

Nous avons vu beaucoup de choses dans ce chapitre :

- une première méthode de tracé de cercle
- des effets graphiques accidentels mais intéressants dûs à quelques bugs

Dans le chapitre suivant, nous allons étudier une autre méthode algorithmique de tracé de cercle, et surtout nous allons enfin réaliser notre compas électronique (depuis le temps que je vous le promets...).

2.8.3 Tracé de cercle – algo n°2 (avec sinus & cosinus)

L'algorithme de tracé de cercle vu dans le chapitre précédent n'est pas le plus pratique qui soit, surtout si l'on souhaite animer le tracé du cercle à la manière d'un compas.

Pour mettre en œuvre l'algorithme qui va suivre, nous avons besoin :

- du nombre PI
- des fonctions trigonométriques « sin » (Sinus) et « cos » (Cosinus)
- des coordonnées XY du centre du cercle, et de son rayon
- et accessoirement de la fonction « radians » dont nous reparlerons

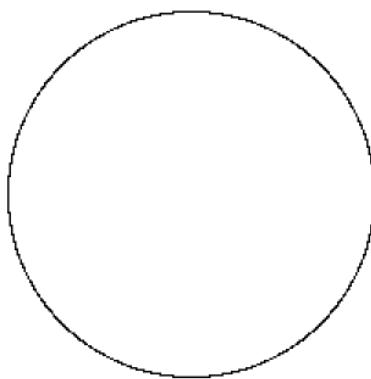
Voici le code source :

```
var rayon = 100;
var pas_increment = .1;

function setup() {
    createCanvas(620, 250);
    noFill();
}

function draw() {
    var i;
    var posx = width/2;
    var posy = height/2;
    for (i = 0 ; i <= 360; i += pas_increment) {
        var angle = radians(i);
        var x = posx + rayon * cos(angle);
        var y = posy + rayon * sin(angle);
        point(x, y);
    }
    if (frameCount > 200) noLoop();
}
```

Résultat obtenu :



D'un point purement mathématique, tracer un cercle consiste ici à calculer un certain nombre d'angles et à tracer les points correspondants. Nous en calculons ici $360/0.1$ soit 3600 angles (cela nous donne une très bonne précision pour le tracé).

Vous remarquerez que chaque angle est converti en radians avant d'être transmis aux fonctions Cosinus (pour le calcul de X) et Sinus (pour le calcul de Y), car ces dernières ne savent travailler que dans cette unité. Au fait, un radian, c'est quoi :

« Le radian, c'est la mesure de l'arc dont la longueur est exactement égale au rayon du cercle qui le définit. »

Si ça vous fait mal aux cheveux, rassurez-vous, ce n'est pas une information essentielle à ce stade (vous pourrez l'approfondir plus tard).

A propos de Sinus et Cosinus, P5 met à notre disposition les fonction « sin » et « cos » qui sont en fait des raccourcis vers les fonctions « Math.sin » et « Math.cos » du langage Javascript. Vous pouvez utiliser « sin » ou « Math.sin » indifféremment (idem pour « cos » et « Math.cos »). L'intérêt d'utiliser « sin » et « cos » réside surtout dans le fait que ces fonctions existent aussi sur Processing, donc vous aurez moins de travail d'adaptation à faire si vous décidez de convertir votre sketch de P5 vers Processing.

Mais revenons à nos moutons. Il me semble qu'il est intéressant de s'arrêter un instant sur la variable « pas_increment ». Augmentez-la, diminuez-la, regardez ce qui se passe. Le tracé est-il plus ou moins précis ? Le calcul est-il plus ou moins rapide ?

Revenons maintenant à notre problématique d'animation du cercle.

Je rappelle que nous avons placé notre algorithme de tracé du cercle dans la fonction draw(), qui est appelée environ 60 fois par seconde. Grâce au test placé à la fin de la fonction « draw » sur la variable « frameCount », nous laissons la fonction « draw » s'exécuter 200 fois avant de la stopper.

OK, mais pourquoi n'y a-t-il pas d'animation ?

Pour comprendre ce qui se passe, j'ai placé les fonctions Javascript console.time() et console.timeEnd(), au début et à la fin de la fonction draw(), dans le sketch de la page précédente. Vous pouvez donc le réafficher avec la console Javascript "ouverte" (cf. touche F12).

```
function draw() {
    console.time('sketch');
    // ...
    console.timeEnd('sketch');
}
```

Voici un extrait de ce qui se trouve dans la console :

```
sketch : minuteur démarré  
sketch : 15.23 ms  
sketch : minuteur démarré  
sketch : 13.55 ms  
sketch : minuteur démarré  
sketch : 15.01 ms
```

Le temps moyen d'exécution se situe autour des 14 milli-secondes. On atteint donc sans problème les 60 FPS (« frame per second ») qui devraient nous permettre d'obtenir une animation fluide. Alors, pourquoi n'a-t-on pas d'animation à l'arrivée ?

Pour comprendre, il faut préciser le fonctionnement de la fonction « draw » : à la fin de chaque appel de cette fonction, ce qui est envoyé au navigateur, c'est le résultat complet de l'image produite par cette fonction. A aucun moment nous n'envoyons au navigateur une version transitoire de ce que la fonction « draw » est en train de générer.

Pour obtenir notre effet d'animation « façon compas », nous devons faire en sorte que la fonction « draw » génère un point du cercle à chaque appel, et non pas la totalité du cercle (ce que nous faisons actuellement).

Nous devons donc reconsidérer le problème et envisager une autre manière de faire.

Vous ne pouvez pas le voir dans ce support, mais le sketch ci-dessous dessine progressivement un cercle, à la façon d'un compas :

```
var i=0, rayon=100, posx=0, posy=0, pas_increment=.5;  
  
function setup() {  
    createCanvas(620, 250);  
    noFill();  
    posx = width/2 ;  
    posy = height/2;  
}  
function draw() {  
    var angle = radians(i);  
    var x = posx + rayon * cos(angle);  
    var y = posy + rayon * sin(angle);  
    point(x, y);  
    i = i + pas_increment;  
    if (i > 360) noLoop();  
}
```

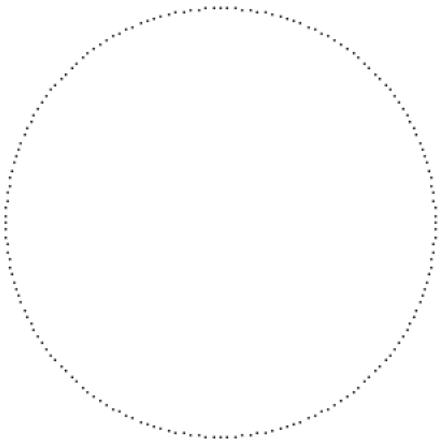
Ce sketch a le mérite de la simplicité, car nous n'avons même pas eu besoin de déclarer de boucle « for ». Nous avons déclaré une variable « i » en dehors de la fonction « draw » (en l'initialisant à zéro) et inséré à la fin de cette fonction « draw » une condition d'arrêt avec le test suivant :

```
if (i > 360) noLoop();
```

En pratique, la fonction « draw » va être exécutée en boucle jusqu'à ce que la valeur de « i » atteigne 360 (en l'occurrence, il s'agit de 360 degrés).

Vous noterez qu'au tout début de la fonction « draw » nous avons converti la variable « i » (exprimée en degrés) en sa valeur équivalente exprimée en radians.

Vous pouvez augmenter la valeur de la variable « pas_increment », pour obtenir des rendus différents. Par exemple, si vous la fixez à 2, vous obtiendrez ceci :



Si vous avez du mal à vous représenter ce qui se passe à l'intérieur de la fonction « draw », je vous invite à insérer la fonction « console.log » dans la fonction « draw » et à afficher le contenu des variables qui vous semblent utiles. Vous pouvez par exemple insérer la ligne en gras ci-dessous :

```
point(x, y);
console.log(frameCount + ' ; i=' + i + ' ; angle=' + angle +
    ' ; x=' + x + ' ; y=' + y);
i = i + pas_increment;
```

Voici un extrait de ce que vous obtiendrez dans la console du navigateur :

```
"1 ; i=0 ; angle=0 ; x=410 ; y=125"
"2 ; i=2 ; angle=0.03490658503988659 ; x=409.93908270190957 ; y=128.4899496702501"
"3 ; i=4 ; angle=0.06981317007977318 ; x=409.7564050259824 ; y=131.97564737441252"
"4 ; i=6 ; angle=0.10471975511965977 ; x=409.45218953682735 ; y=135.45284632676535"
"5 ; i=8 ; angle=0.13962634015954636 ; x=409.02680687415705 ; y=138.91731009600653"
...
"177 ; i=352 ; angle=6.14355896702004 ; x=409.02680687415705 ; y=111.08268990399341"
"178 ; i=354 ; angle=6.178465552059927 ; x=409.45218953682735 ; y=114.54715367323466"
"179 ; i=356 ; angle=6.213372137099814 ; x=409.7564050259824 ; y=118.02435262558753"
"180 ; i=358 ; angle=6.2482787221397 ; x=409.93908270190957 ; y=121.51005032974992"
"181 ; i=360 ; angle=6.283185307179586 ; x=410 ; y=124.99999999999997"
```

On voit donc que la fonction « draw » s'est exécutée 181 fois, et que la variable « i » est passée graduellement de 0 à 360 degrés, avec un pas de 2 (car c'est la valeur que j'avais définie dans la variable « pas_increment »).

Je rappelle que les coordonnées cartésiennes calculées pour chaque point du cercle, contiennent des décimales du fait de la nature des calculs effectués. On a vu précédemment que la fonction « point » s'en débrouillait très bien, puisqu'elle tronque la partie décimale et conserve la partie entière de chaque coordonnée. Mais cela peut quelquefois créer des lignes et des arrondis disgracieux. On peut réduire ces effets disgracieux en utilisant la fonction « round », qui a pour effet d'arrondir les nombres à l'entier le plus proche. On en a un exemple ci-dessous :

```
point(round(x) , round(y)) ;
```

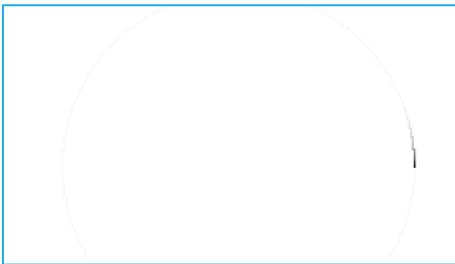
2.8.4 Compas avec effet de traîne

J'aimerais maintenant tracer un cercle, à la façon d'un compa, mais un compas avec une encre magique, qui s'effacerait au bout de quelques secondes. Je voudrais obtenir un effet de traîne, en quelque sorte.

Il existe une première manière très simple pour obtenir cet effet. Elle consiste à utiliser une technique que nous avons déjà vue, la fonction « background » associée à une notion d'opacité. Par exemple si l'on reprend le sketch du chapitre précédent, que l'on ajoute la ligne suivante au tout début de la fonction « draw » :

```
background(255, 20);
```

On obtient rapidement l'effet recherché, avec un cercle qui se dessine, et dont les points les plus anciens disparaissent progressivement :



L'effet n'est pas mal, mais si vous jouez sur la valeur du second paramètre de la fonction « background » (définissant l'opacité), vous constaterez que n'avez pas un contrôle très précis concernant l'effet obtenu. Imaginez que vous souhaitez combiner plusieurs éléments sur l'écran, mais que vous souhaitez que ces éléments ne

disparaissent pas tous de la même manière, ou au même rythme, vous allez vite être confronté à un problème.

Il nous faut donc envisager une autre solution, et la solution que j'ai en tête passe par l'utilisation de tableaux. Un tableau en JS, et dans la plupart des autres langages, c'est une collection de valeurs stockées sur le même emplacement.

J'évoquais précédemment l'idée d'une grande armoire pour représenter la mémoire de l'ordinateur, et vous indiquais d'imaginer chaque variable comme un tiroir de cette grande armoire. Eh bien, un tableau, c'est aussi un tiroir de cette grande armoire, mais un tiroir qui va lui-même contenir une autre armoire, avec sa propre série de tiroirs.

Certains langages imposent que tous les éléments du tableau soient de même type (c'est le cas de Java), mais pas Javascript qui est beaucoup plus souple.

En JS, on peut initialiser un tableau vide de cette manière :

```
var datas = [] ;
```

Seconde manière quasi-équivalente :

```
var datas = new Array() ;
```

Il y a une subtile différence, entre ces 2 méthodes d'initialisation de tableau, mais à notre niveau et pour ce que nous allons faire ici, cela ne fera pas de réelle différence, aussi je préfère ne pas m'attarder sur le sujet, et je vous recommande d'utiliser la 1^{ère} méthode.

Pour ajouter des éléments dans un tableau, on peut utiliser la méthode « `push` » qui est associée – en Javascript – à toute variable de type tableau :

```
var datas = [] ; // création d'un tableau vide
datas.push('ceci est une donnée') ; // premier poste de tableau
datas.push('ceci est autre une autre donnée') ; // second poste de tableau
datas.length; // ça nous donne le nombre de postes du tableau
```

Si vous exécutez ce bout de code dans la console de Google Chrome, cela donne :

```
> var datas = [] ; // création d'un tableau vide
  datas.push('ceci est une donnée') ; // premier poste de tableau
  datas.push('ceci est autre une autre donnée') ; // second poste de tableau
  datas.length; // ça nous donne le nombre de postes du tableau
< 2
>
```

Le navigateur nous renvoie la valeur « 2 » ce qui correspond au nombre d'éléments du tableau « datas » que nous venons de créer et d'alimenter. Vous entendrez parfois parler des « postes » d'un tableau, parfois des « éléments » d'un tableau, ces deux termes sont synonymes.

En ce qui nous concerne, nous aurons besoin de stocker une liste de coordonnées cartésiennes. Nous avons pour ce faire deux possibilités :

- créer un tableau des coordonnées X, et en parallèle un tableau des coordonnées Y, doit deux tableaux à gérer
- créer un tableau unique contenant, pour chaque élément du tableau, un jeu de coordonnées X et Y

C'est la seconde solution que j'ai adoptée dans l'exemple fictif ci-dessous :

C'est la seconde solution que j'ai adoptée dans l'exemple fictif suivant :

```
var datas = [] ; // tableau de coordonnées vide
for (var i = 0; i <= 100; i+= 1) {
    var j = i * 1.5;
    datas.push({x:i, y:j});
}
```

Dans cette petite boucle, pour toutes les valeurs de i allant de 0 à 100 inclus (avec un pas de 1), nous calculons les valeurs de j (la formule est bidon, n'y attachez pas d'importance). Ce qui nous intéresse ici, c'est la manière dont nous stockons les valeurs de i et j :

```
datas.push({x:i, y:j});
```

Ce qui se trouve entre les accolades {}, c'est la description d'un objet Javascript. Nous créons ici un objet auquel nous ne donnons pas de nom, et nous lui ajoutons à la volée deux propriétés x et y, auxquelles nous transmettons les valeurs des variables i et j.

C'est quoi une propriété d'objet ? Eh bien, c'est une variable, mais une variable qui appartient à cet objet. Dans notre exemple, nous n'avons que deux propriétés x et y, mais nous pouvons ajouter beaucoup d'autres. Si l'on repense à l'image de l'armoire, la liste des propriétés d'un objet, c'est une grande armoire qui appartient à cet objet, avec ses propres tiroirs bien évidemment.

Il existe d'autres manières d'écrire le même objet. L'une de ces autres manières, qui est fréquemment utilisée par les développeurs, c'est celle-ci-dessous :

```
var objet_temp = {} ; // création d'un objet vide
objet_temp.x = i ; // ajout propriété x à l'objet, et stockage de la valeur de i
objet_temp.y = j ; // ajout propriété y à l'objet, et stockage de la valeur de j
datas.push(objet_temp) ; // stockage de l'objet dans le tableau datas
```

Le code ci-dessus est strictement équivalent au code ci-dessous, d'un point de vue du résultat final :

```
datas.push({x:i, y:j});
```

Dans quel cas utiliser la forme « raccourcie », dans quel cas utiliser la forme plus verbeuse ? Il n'y a pas de règle stricte, je dirais qu'en général je privilégie la forme raccourcie quand j'ai peu de données à injecter dans l'objet. C'est le cas ici, car je n'ai que deux propriétés x et y. Quand la liste des propriétés est plus importante, j'ai tendance à privilégier la forme plus verbeuse, pour des raisons de lisibilité. Mais il m'arrive aussi de préférer la forme raccourcie, et de simplement déclarer chaque propriété sur des lignes distinctes, comme dans l'exemple ci-dessous, qui est strictement équivalent à l'exemple précédent :

```
datas.push({
    x:i,
    y:j
});
```

Fort de ces connaissances nouvelles, il est temps de revenir à notre problème de génération de graphique. Je rappelle que notre objectif est de précalculer les coordonnées cartésiennes de tous les points de notre graphique, de stocker toutes ces coordonnées dans un tableau, et d'exploiter le contenu de ce tableau pour dessiner la forme de manière progressive.

Pour alimenter notre tableau de coordonnées, nous utiliserons un code ayant grossomodo la forme suivante :

```
var datas = [] ; // tableau de coordonnées vide
var ray = 100 ; // rayon du cercle
for (var i = 0; i <= 360; i+= .5) {
    var angle = radians(i);
    var x = ray * Math.cos(angle);
    var y = ray * Math.sin(angle);
    datas.push({x:x, y:y});
}
```

Dans le sketch suivant, nous précalculons les coordonnées cartésiennes des différents points définissant notre cercle, et nous les stockons dans un tableau que j'ai appelé « data ». Nous utilisons ensuite le contenu de ce tableau, pour afficher un pixel pour chaque nouvel élément de ce tableau, ceci à chaque exécution de la fonction « draw » :

```
// précalcul du cercle et stockage dans un tableau
var x, y, i, imax, ray, pas, posx, posy;
var data = [];
var ray = 100;
var i, imax = 0;
var pas = .5;
```

```

function setup() {
    createCanvas(620, 250);
    noFill();
    stroke('black');
    posx = width/2;
    posy = height/2;
    for (i = 0; i <= 360; i+= pas) {
        var angle = radians(i);
        x = posx + ray * cos(angle);
        y = posy + ray * sin(angle);
        data.push({x:x, y:y});
    }
    i = 0 ;
    imax = data.length;
}
function draw() {
    if (i < imax) {
        point(data[i].x, data[i].y);
        i += 1;
    } else {
        noLoop();
    }
}

```

Le sketch de la page précédente dessine bien notre cercle à la manière d'un compas, mais il lui manque l'effet de traîne que je souhaitais obtenir.

Pour obtenir cet effet de traîne, nous allons modifier très légèrement notre sketch : pour chaque point "dessiné", nous allons afficher les 5 points précédents avec un effet de dégradé (plus on s'éloigne du point de référence, et plus la couleur s'éclaircit).

```

// boucle de tracé des 5 points "suiveurs" à placer après
// le tracé du point principal
for (var j = i-1 , k=0; j >= 0 && k<=5 ; j -= 1, k--) {
    col += 5; // effet de dégradé pour les points "suiveurs"
    stroke(col);
    point(data[j].x, data[j].y); // tracé des points "suiveurs"
}

```

Notre fonction « draw » modifiée se présentera de la façon suivante :

```

function draw() {
    var col = 0; // premier point en noir
    stroke(col);
    if (i < imax) {
        point(data[i].x, data[i].y); // le point principal (le "guide")
        i += 1;
        // boucle de tracé des 5 points "suiveurs"
        for (var j = i-1 , k=0; j >= 0 && k<=5 ; j -= 1, k--) {
            col += 5; // effet de dégradé pour les points "suiveurs"
            stroke(col);
            point(data[j].x, data[j].y); // tracé des points "suiveurs"
        }
    }
}

```

```
    } else {
        noLoop();
    }
}
```

Appliquez ces modifications et relancez le sketch, vous allez obtenir un effet de traîne que vous allez pouvoir contrôler complètement.

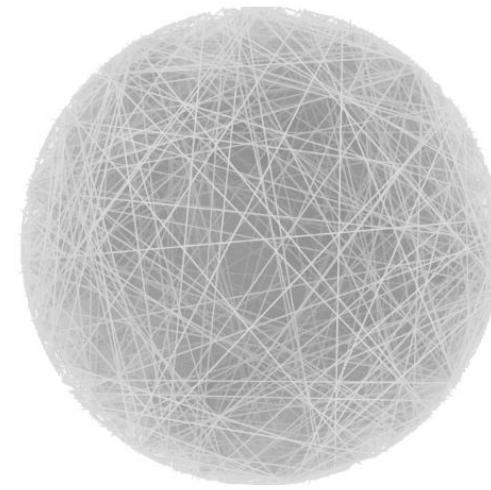
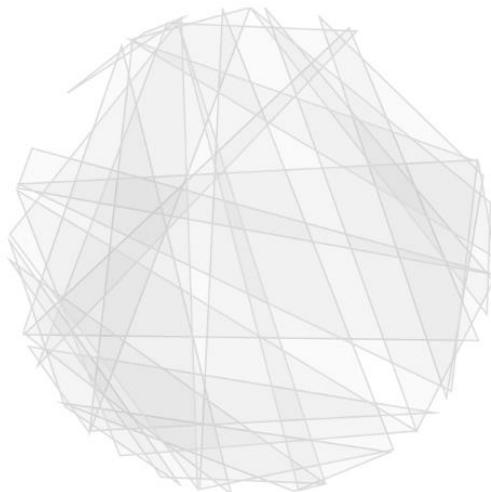
Vous trouverez en annexe un chapitre d'introduction aux tableaux, qui vous permettra de compléter vos connaissances sur ce sujet spécifique.

2.8.5 Cercle en folie

Nous allons voir un autre exemple d'utilisation du tableau des coordonnées précalculées.

Le sketch qui suit va sélectionner dans le tableau « data », au hasard, 3 jeux de coordonnées cartésiennes qui vont constituer les sommets d'un triangle.

En répétant la technique un grand nombre de fois, avec un effet de transparence en prime, on obtient un effet sympa. Je vous donne ci-dessous un aperçu, à deux étapes différentes de la construction de l'image :



Vous trouverez page suivante le code source du sketch. Prenez le temps qu'il faut pour l'analyser, et bien le comprendre. Vous pourrez le modifier pour - par exemple - générer des parallélépipèdes plutôt que des triangles, en vous appuyant sur la fonction « quad » dont la documentation se trouve ici :

<http://p5js.org/reference/#/p5/quad>

```

// Triangles aléatoires inscrits dans un cercle (version P5)
var x, y, posx, posy;
var data = [];
var ray = 150;
var i, imax = 0;
var pas = .5;

function point_cercle(posx, posy, ray, angle) {
    var x,y;
    x=posx+ray*cos(angle);
    y=posy+ray*sin(angle);
    data.push({x:x, y:y});
}

function setup() {
    createCanvas(620, 300);
    fill(120, 10);
    stroke(220);

    posx = width/2;
    posy = height/2;

    // précalcul du cercle pour stockage dans le tableau "data"
    for (i = 0; i <= 360; i+= pas) {
        point_cercle(posx, posy, ray, radians(i));
    }
    i = 0 ;
    imax = data.length-1;

    frameRate(20); // animation pas trop rapide pour avoir le temps de la voir
}

function draw() {
    var coords = [];
    var tmppos ;
    tmppos = int(random(0, imax));
    coords.push({x:data[tmppos].x, y:data[tmppos].y});
    tmppos = int(random(0, imax));
    coords.push({x:data[tmppos].x, y:data[tmppos].y});
    tmppos = int(random(0, imax));
    coords.push({x:data[tmppos].x, y:data[tmppos].y});

    triangle( coords[0].x, coords[0].y,
              coords[1].x, coords[1].y,
              coords[2].x, coords[2].y);

    if (frameCount > 200) noLoop();
}

```

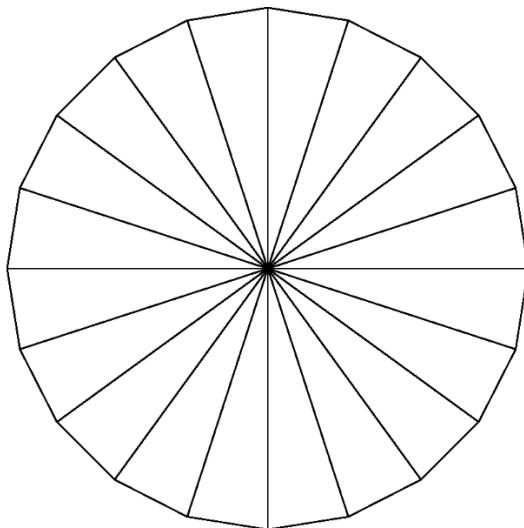
2.8.6 Polygones

Voici un petit sketch qui dessine des polygones à l'écran :

```
var sx = 600;
var sy = 500;
var backcolor = "white";
var drawcolor = "black";
var xc, yc;
var rayon = 200;
var s = 20; // nombre de côtés du polygone

function setup() {
    createCanvas(sx, sy);
    background(backcolor);
    stroke(drawcolor);
    xc = sx / 2;
    yc = sy / 2;
}
function draw() {
    for (var a = 0 ; a <= 360 ; a+= 360/s) {
        var x = xc + rayon * cos(radians(a));
        var y = yc + rayon * sin(radians(a));
        if (a != 0) {
            line(px, py, x, y);
            line(x, y, xc, yc);
        }
        var px = x;
        var py = y;
    }
}
```

Résultat obtenu :



Fort de l'expérience acquise dans les chapitres précédents, je propose que vous essayiez d'animer la construction de ce polygone, en appliquant la technique étudiée au chapitre 2.8.4 (avec précalcul et stockage des coordonnées cartésiennes dans un tableau).

Vous trouverez la solution à ce problème en annexe (cf. chapitre 5.2).

3. Partir à l'aventure

Nous allons aborder de nouvelles techniques, tout en nous appuyant sur ce que nous avons appris dans les chapitres précédents. Vous allez découvrir de nouveaux sketchs. Vous aurez tout le loisir de les saisir et de les tester directement, puis de lire les explications qui les accompagnent. Vous aurez aussi la possibilité de travailler de manière plus graduelle, en mode « pas à pas » en quelque sorte. La manière importe peu, ce qui compte c'est que vous puissiez vous approprier les techniques que nous allons voir, de manière à pouvoir les « hacker », c'est-à-dire les bidouiller, les exploiter pour explorer de nouveaux territoires.

3.1 Courbes polaires

Dans ce chapitre, nous allons aborder un nouveau sujet, les courbes polaires, ou plus exactement, les courbes en coordonnées polaires. Pour dessiner ces courbes, nous allons utiliser des fonctions polaires, ce qui va nous donner l'occasion de créer nos propres fonctions.

Je crois important de souligner que j'ai tiré – pour rédiger ce chapitre - beaucoup d'éléments d'un petit livre écrit par Czes Kosniowski :

« *Fun Mathematics on your microcomputer* » (Cambridge University Press 1983)

Je n'ai découvert que récemment ce petit livre à la couverture rouge. Je l'ai trouvé passionnant, du coup cela m'a amené à faire quelques recherches sur son auteur, et j'ai découvert que ce professeur de maths britannique avait, dans les années 80, écrit plusieurs petits livres d'initiation à la programmation (pour Commodore 64 et Oric Atmos notamment), ainsi qu'un livre pour apprendre à maîtriser le Rubik's Cube.

Dans le livre « *Fun mathematics on your microcomputer* », Czes Kosniowski propose plusieurs exemples de programmes écrits en langage BASIC, dont un pour tracer ces fameuses courbes polaires.

3.1.1 Abeilles polaires

Dans le livre précité, Czes Kosniowski explique ceci :

« *Une fonction qui utilise des coordonnées polaires (R,Z) est appelée une fonction polaire. Par exemple, $R=\sin(Z)$ est une fonction polaire. Le point (R,Z) en coordonnées polaires est le même que le point (X,Y) qui a pour coordonnées cartésiennes :*

$$R*\cos(Z), R*\sin(Z).$$

Ainsi, le graphe de la fonction polaire $R=\text{Sin}(Z)$ est très différent de la fonction classique $Y=\text{Sin}(X)$. »

En fait, un graphe polaire, c'est un cercle... qui serait frappé d'amnésie et ne saurait plus trop où il habite, ou si préférez une vision plus romantique... un cercle qui chercherait à échapper à sa condition de cercle 😊. Bon, ce n'est pas de Czes, c'est juste moi qui me fais un délire 😊.

Voici un autre extrait du livre que j'aime beaucoup (traduction perfectible assurée par votre serviteur) :

« Les abeilles utilisent des coordonnées polaires pour communiquer des informations sur les sources de nourriture. Après avoir trouvé une nouvelle source (un parterre de fleurs), une abeille éclaireuse retourne à la ruche, dépose un échantillon de la nourriture et exécute une danse pour montrer où se trouve cette nourriture... »

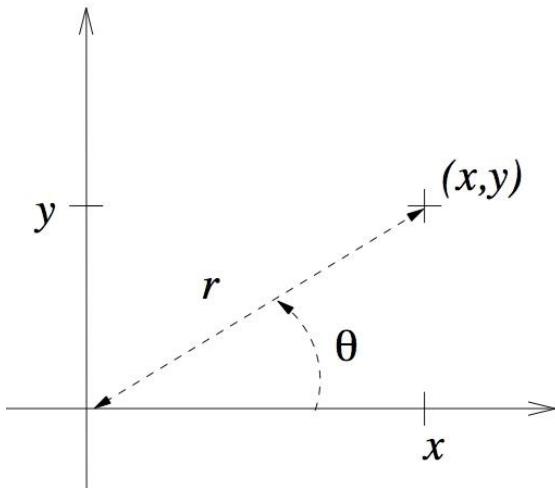
Dans la suite de son explication, Czes nous apprend que les abeilles, en effectuant cette danse de la nourriture, dessinent des courbes polaires. Intrigué, je lance une petite recherche sur internet, et je découvre que ces faits sont connus depuis fort longtemps, et que le monsieur qui a mis en évidence cette danse des abeilles s'appelait Karl von Frisch (1886 - 1982), éthologue mondialement connu et Prix Nobel de médecine 1973. Si vous avez envie d'en savoir davantage :

https://fr.wikipedia.org/wiki/Karl_von_Frisch

Si vous n'êtes pas sûr d'avoir compris ce que sont les coordonnées polaires, la définition proposée par Wikipédia vous éclairera peut être :

*Chaque point du plan est déterminé par les **coordonnées polaires**, qui sont la **coordonnée radiale** et la **coordonnée angulaire**. La **coordonnée radiale** (souvent notée r ou ρ , et appelé **rayon**) exprime la distance du point à un point central appelé **pôle** (équivalent à l'origine des **coordonnées cartésiennes**).*

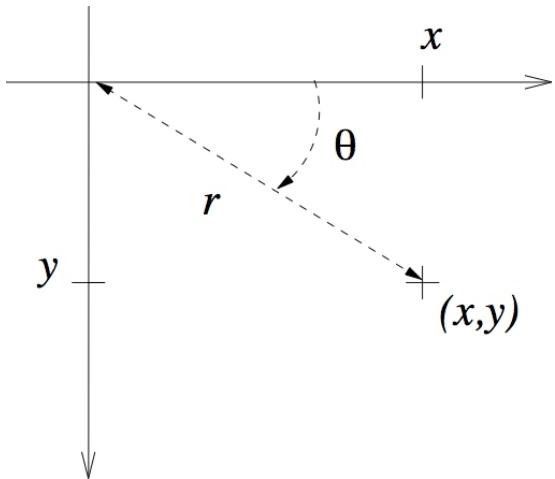
Ce graphique pris sur Wikipédia est assez éloquent :



Donc un point de coordonnées cartésiennes XY est représenté en coordonnées polaires par un rayon et un angle. On en revient donc à la fameuse formule proposée par Czes que j'ai placée au début de ce chapitre :

$$R * \text{Cos}(Z), R * \text{Sin}(Z).$$

Je rappelle que sur l'écran de l'ordinateur, le coin supérieur gauche correspond au point de coordonnées cartésiennes 0,0, et que l'axe vertical est inversé, aussi le graphe ci-dessus doit plutôt être regardé de la façon suivante (d'un point de vue informatique) :



Czes propose dans son livre un programme, écrit en langage BASIC, qui permet de tracer des courbes polaires, et il propose un certain nombre de fonctions mathématiques à explorer via ce programme. En voici quelques unes :

```
z * z * sin(1/z);
sin(2 * z);
```

```

sin(7 * z);
1 + 2 * cos(z);
1 + cos(z);
1 + sin(2 * z);
1 + 2 * cos(2*z);
sin(5 * z);
sin(6 * z);
sin(8 * z);

```

J'ai eu envie de convertir le programme BASIC de Czes en Javascript, en partie pour le côté « fun » de l'exercice, et surtout parce que j'étais curieux de voir ce que ces fonctions allaient donner avec la résolution graphique des ordinateurs d'aujourd'hui.

La lecture du programme BASIC m'a amené à considérer que j'allais avoir besoin de quelques variables globales. Dans les vieux programmes BASIC des années 80, les notions de variables globales et locales n'existaient pas, toute variable était globale. Pour me simplifier la vie, j'ai dans un premier temps défini toutes les variables comme étant globales (donc en dehors des fonctions « setup » et « draw »), et petit à petit j'ai élagué pour mieux structurer mon sketch. Du coup, j'ai repassé certaines variables en local, après avoir considéré que c'était plus pertinent dans certains cas. Si vous avez du mal à voir quand une variable doit être globale ou locale, essayez d'appliquer le principe suivant :

- les variables dont le contenu est utile à plusieurs fonctions doivent être globales
- les variables dont le contenu est utile à une et une seule fonction doivent être définies localement

Voici la liste des variables globales que j'ai retenues dans la version finale du sketch :

```

"use strict";
var sx = 800;
var sy = 500;
var points = [];
var points_max = 0;
var points_i = 0;
var precision = 0.001;

```

La variable « precision » aurait pu être définie comme variable locale de la fonction « setup », mais je l'ai mise ici volontairement pour pouvoir la retrouver et la modifier plus facilement.

Tiens, c'est quoi ce « use strict » que j'ai placé au tout début du sketch, juste avant mes variables globales ? Il s'agit d'un indicateur JS qui permet de demander à l'interpréteur Javascript de se comporter de manière plus stricte... mais plus stricte par rapport à quoi ?

Je vous ai dit que dans le langage JS, vous pouvez créer des variables locales et globales. Pour déclarer une variable, on a vu qu'il fallait utiliser le mot clé « var ». Mais il arrive que l'on oublie d'utiliser ce mot clé « var ». Au lieu de vous prévenir du fait que vous avez écrit une bêtise, l'interpréteur JS considère que votre variable est une variable globale. Quelquefois ce n'est pas trop grave, mais il peut arriver que cette décision arbitraire de l'interpréteur JS vous amène à commettre des erreurs. Par exemple, vous décidez de modifier un bout de code JS et de transférer ce code dans une fonction. Vous pensiez que toutes les variables utilisées dans cette fonction étaient locales, mais du fait que le mot clé « var » a été oublié sur l'une des variables, cette dernière est considérée comme une variable globale, ce qui peut entraîner des bugs potentiels difficiles à repérer (cette variable pourrait par exemple impacter le contenu d'une autre variable de même nom qui se trouve à un niveau supérieur, et est destinée à un autre usage). Ce genre de mésaventure est arrivée à plus d'un développeur, moi y compris.

Heureusement, l'indicateur « use strict » permet de modifier le comportement par défaut de l'interpréteur JS, et de lui dire qu'on ne tolère pas l'utilisation de variables non déclarées. Dans ce contexte, dès que l'interpréteur rencontre une variable qui est utilisée - par exemple dans un calcul - alors qu'elle n'a pas été déclarée au préalable par le mot clé « var », il s'arrête en renvoyant un message d'erreur. Ce message d'erreur, vous le retrouverez dans la console du navigateur, il ressemble à ceci :

 ReferenceError: assignment to undeclared variable points_i [En savoir plus]

A droite du message d'erreur ci-dessus, vous trouverez en bout de ligne le nom du fichier dans lequel l'erreur a été détectée (ici il s'agit du fichier « czes_v1.js »). Le chiffre « 6 » désigne le numéro où l'erreur s'est produite. Le chiffre « 1 » indique que l'erreur a été détectée sur le premier caractère de la ligne « 6 » :

...  czes_v1.js:6:1

J'ai pris l'habitude d'utiliser l'indicateur « use strict » de manière systématique au début de chacun de mes sketchs. Quand j'oublie de le faire, je m'en mords les doigts presque à chaque fois. Je pense que très vite, et avec un peu de pratique, vous conviendrez que ce mode strict est bien pratique.

Je vous disais que Czes proposait dans son livre un certain nombre de fonctions mathématiques à tester. Je me suis dit que, dans un premier temps, le plus simple était de définir la liste de ces fonctions en commentaire, et de les tester au fil de l'eau en commentant les fonctions que je ne voulais pas exécuter, et en décommentant celle que je souhaitais tester.

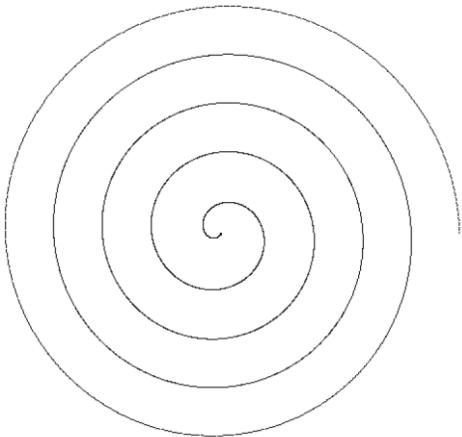
Dans son programme en BASIC, Czes avait nommé « fna » sa fonction polaire. Je pourrais l'appeler « toto » ou encore « polarfunc », cela a peu d'importance, aussi je vais conserver le nom de « fna ».

Voici donc la fonction « fna », dans une de ses toutes premières versions. Nous verrons par la suite qu'il y a des manières plus élégantes et pratiques de procéder :

```
var fna = function(z) {
    // return z * z * sin(1/z);
    // return 1 ;
    // return sin(2 * z);
    // return sin(7 * z);
    // return 1 + 2 * cos(z);
    // return 1 + cos(z);
    // return 1 + sin(2 * z);
    return 1 + 2 * cos(2*z); // c'est la fonction active du moment :)
    // return sin(5 * z);
    // return sin(6 * z);
    // return sin(8 * z);
}
```

Le début de la fonction « setup » est sans surprise, mais les variables « a » et « b » peuvent vous surprendre. Ces deux variables ont un effet de « modulation » sur la courbe générée. Dans un premier temps, je les fixe arbitrairement à 5, vous pourrez vous amuser à tester la même fonction polaire avec des valeurs différentes pour « a » et « b ».

Voici une jolie spirale obtenue – après de nombreux essais - avec les variables « a » et « b » fixées toutes deux à « 5 », et la fonction suivante : $z * z * \sin(1/z)$



Dans la suite de la fonction « setup », on trouve deux boucles de calculs et de stockage de coordonnées, sur lesquelles je reviendrai après vous avoir fourni l'intégralité du code :

```
"use strict";
var sx = 800;
var sy = 500;
var points = [];
var points_max = 0;
var points_i = 0;
var precision = 0.001;

var fna = function(z) {
    return z * z * sin(1/z);
```

```

// return 1 ;
// return sin(2 * z);
// return sin(7 * z);
// return 1 + 2 * cos(z);
// return 1 + cos(z);
// return 1 + sin(2 * z);
// return 1 + 2 * cos(2*z); // c'est la fonction active du moment :]
// return sin(5 * z);
// return sin(6 * z);
// return sin(8 * z);
}

function setup() {
  createCanvas(sx, sy);
  stroke('black');

  // détermination des coordonnées du centre du canvas
  var hx = sx / 2;
  var hy = sy / 2;

  var a = 5;
  var b = 5;

  // 1ère boucle : pour obtenir une approximation
  // de la plus grande valeur du rayon "r" pouvant
  // être obtenue en cours de calcul
  var m = 1.0e-30;
  var r, z;
  for (z=0 ; z <= TWO_PI ; z+=.1) {
    r = abs(fna(z));
    if (m < r) {
      m = r + 0.1;
    }
  }

  // 2ème boucle : calcul des coordonnées XY
  // et stockage dans le tableau "points"
  var u, v;
  for (z=0 ; z <= TWO_PI ; z+= precision) {
    r = abs(fna(z));
    u = hx + hy * cos(a * z) * r / m;
    v = hy + hy * sin(b * z) * r / m ;
    if (v < 0 || v > sy) {
      console.log('nope');
    } else {
      points.push({x:u, y:v});
    }
  }
  points_max = points.length;
  points_i = 0;
}

```

```

function draw() {
    // parcours et tracé de l'ensemble des points du tableau "points"
    point(points[points_i].x, points[points_i].y);
    points_i++;
    if (points_i >= points_max) {
        // si on atteint le dernier élément du tableau "points"
        // ... sauvegarde de la dernière image
        // (ce n'est pas obligé, c'est juste une proposition)
        save("tst-" + frameCount + ".png", 'png');
        // ... arrêt de la boucle sur la fonction "draw"
        noLoop();
    }
}

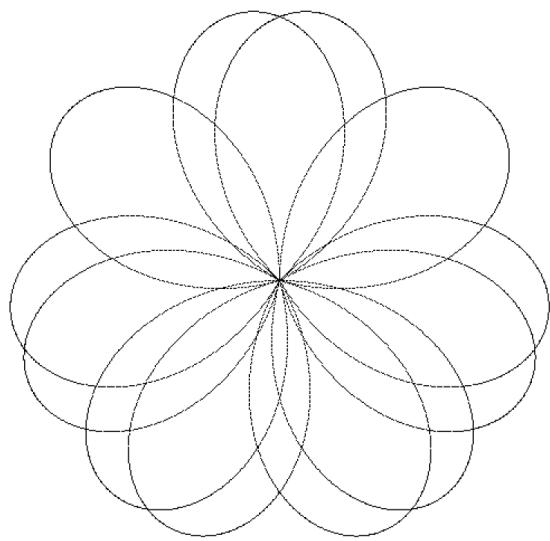
function keyPressed() {
    if (key == 'x' || key == 'X') {
        // possibilité d'arrêter la fonction "draw" à tout moment
        noLoop();
    } else {
        if (key == 's' || key == 'S') {
            // sauvegarde de l'image en cours
            save("tst-" + frameCount + ".png", 'png');
        }
    }
}

```

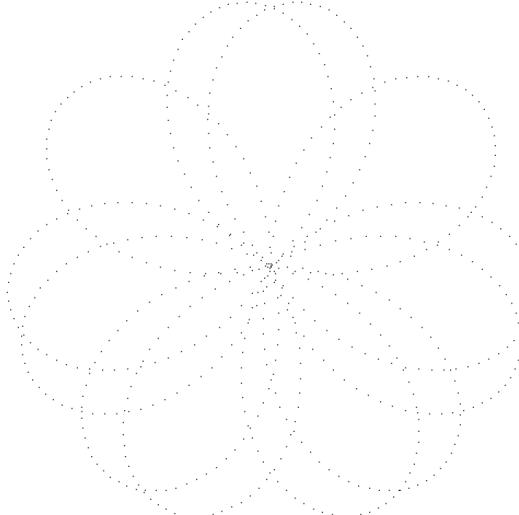
Voici les images que j'ai obtenues pour quelques fonctions polaires :

Pour la fonction : $\sin(6 * z)$

precision : 0.001

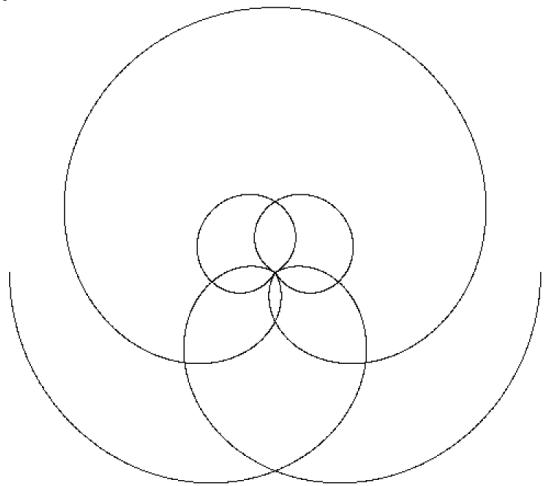


precision : 0.01

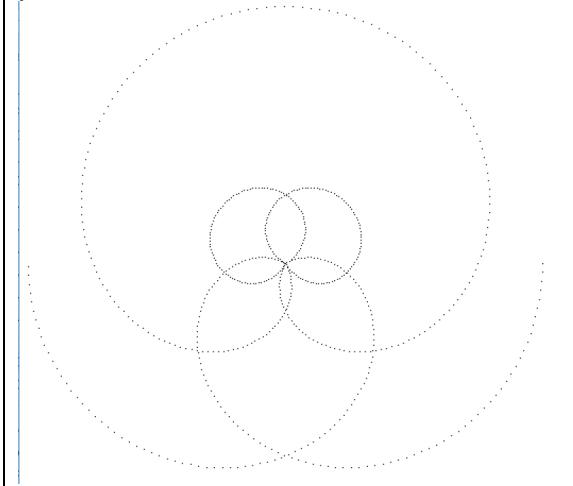


Pour la fonction : $1 + 2 * \cos(2*z)$

precision : 0.001



precision : 0.01



Je vous disais précédemment que la fonction « setup » contenait deux boucles de calcul. Si le rôle de la seconde boucle est assez clair (elle sert à calculer les différents points de la courbe polaire), en revanche le rôle de la première boucle a de quoi intriguer.

On a en premier lieu une variable « m » initialisée avec une valeur infinitésimale. On calcule ensuite la variable « r » pour tous les points du cercle allant de zéro à 2*PI. Le pas d'incrémentation de la boucle est de valeur 0.1 pour la première boucle, alors qu'il est de 0.001 pour la seconde boucle. Cela nous permet de déterminer la valeur la plus élevée de « r ». La valeur la plus élevée de « r » est stockée dans la variable « m », en remplacement de sa valeur précédente. Nous utiliserons ensuite cette variable « m » dans la seconde boucle, comme ratio permettant de « réduire » les calculs de manière à ce qu'ils demeurent circonscrits dans le canvas :

```
// 1ère boucle : pour obtenir une approximation
// de la plus grande valeur du rayon "r" pouvant
// être obtenue en cours de calcul
var m = 1.0e-30;
var r, z;
for (z=0 ; z <= TWO_PI ; z+=.1) {
    r = abs(fna(z));
    if (m < r) {
        m = r + 0.1;
    }
}

// 2ème boucle : calcul des coordonnées XY
// et stockage dans le tableau "points"
var u, v;
for (z=0 ; z <= TWO_PI ; z+= precision) {
    r = abs(fna(z));
    u = hx + hy * cos(a * z) * r / m;
    v = hy + hy * sin(b * z) * r / m ;
    if (v < 0 || v > sy) {
        console.log('nope');
    } else {
        points.push({x:u, y:v});
    }
}
```

Que se passerait-il si on supprimait cette première boucle ? Eh bien, faites l'essai, tout simplement en forçant la valeur de « m » à 1 juste après la première boucle (ainsi vous annulez l'effet de cette première boucle).

Vous noterez que les 2 boucles ci-dessus auraient pu être placées dans une fonction distincte, par exemple une fonction « precalcul ». Cela permettrait d'alléger le code de la fonction « setup » qui à ce stade peut sembler un peu confus. On touche là à des questions de lisibilité, de modularité et de maintenabilité du code.

Nous avons vu beaucoup de choses dans ce chapitre :

- le principe des fonctions polaires et comment les tracer à l'écran
- la directive « `use strict` » pour ne pas autoriser l'utilisation de variables non déclarées

3.1.2 Abeilles survitaminées

Maintenant que l'on sait que le sketch fonctionne, je vous propose d'explorer quelques pistes pour l'améliorer.

Vous vous souvenez sans doute de la fonction « fna », elle n'était pas terrible pas vrai ? Si on souhaitait utiliser une fonction plutôt qu'une autre, nous étions obligés de modifier le code de la fonction à chaque fois, c'était moche... 😞. Ce serait bien de définir une liste de fonctions, et de pouvoir utiliser l'une ou l'autre de ces fonctions à la demande, sans voir à modifier des lignes de code. On pourrait même laisser notre sketch sélectionner une fonction au hasard dans cette liste.

Une manière pratique de stocker cette liste de fonctions, c'est de définir un tableau. J'ai appelé mon tableau « fnaContainer ». Je stocke dans ce tableau les différentes fonctions, sous la forme de simples chaînes de caractères :

```
var fnaContainer = [];
fnaContainer.push('z * z * sin(1/z)');
fnaContainer.push('1');
fnaContainer.push('sin(z)');
fnaContainer.push('sin(2 * z)');
fnaContainer.push('sin(7 * z)');
fnaContainer.push('1 + 2 * cos(z)');
fnaContainer.push('1 + cos(z)');
fnaContainer.push('1 + sin(2 * z)');
fnaContainer.push('1 + 2 * cos(4*z)');
fnaContainer.push('sin(5 * z)');
fnaContainer.push('sin(6 * z)');
fnaContainer.push('sin(8 * z)');
fnaContainer.push('cos(z) * sin(2*z)');
var fnaActive = '';
```

Du coup, je n'ai plus besoin de la fonction « fna ». Je peux la supprimer de la liste des variables globales.

En revanche, j'ai créé une variable globale « fnaActive », elle me permettra de conserver le code de la fonction que je suis en train d'utiliser, et de l'afficher via la fonction « draw » au moment où j'en aurai besoin.

OK, voyons maintenant quelles modifications nous pouvons effectuer à l'intérieur de la fonction « setup ».

En premier lieu, nous souhaitons sélectionner une fonction au hasard dans la liste des fonctions contenues dans le tableau fnaContainer. Pour ce faire, nous allons procéder en 3 temps :

1. récupérer une valeur entière prise au hasard entre 0 et le nombre maximal d'éléments du tableau – 1 (-1 car le premier poste du tableau porte le numéro zéro)
2. grâce à l'indice de tableau récupéré à l'étape 1, on va extraire du tableau l'élément correspondant dans le tableau fnaContainer, et donc récupérer une chaîne de caractères contenant le code d'une fonction
3. générer dynamiquement la fonction « fna » à partir du code de la fonction récupérée à l'étape 2

Si l'on traduit ces 3 étapes en JS, on obtient le code suivant :

```
var icontainer = int(random(0, fnaContainer.length-1));
fnaActive = fnaContainer[icontainer];
var fna = eval('fna = function(z) { return ' + fnaActive + ' }');
```

La dernière ligne doit sûrement vous interroger. En effet la fonction « eval » est une fonction JS que nous n'avions pas encore vu. Concrètement, nous passons à cette fonction une chaîne de caractères contenant du code JS à exécuter. Si ce code est valide, l'interpréteur JS va l'exécuter. En l'occurrence, notre code consiste à créer une fonction JS qui s'appellera « fna ».

En supposant que la variable « icontainer » contienne la valeur zéro, alors la variable « fnaActive » contiendrait la valeur suivante :

$z * z * \sin(1/z)$

Donc la fonction « eval » va exécuter l'instruction suivante :

```
fna = function(z) { return z * z * sin(1/z) };
```

Observez un détail amusant : au départ nous créons une variable « fna » à laquelle nous affectons le résultat d'une fonction « eval »

```
var fna = eval(...);
```

Cette fonction « eval » exécute une instruction qui a pour effet de recréer la variable « fna » avec un contenu très différent de ce qu'il était au départ. C'est fort, non ? Si l'on y réfléchit deux minutes, cette possibilité que Javascript a de créer (ou recréer) à la volée des portions de code en fonction de nouvelles données est très puissante et assez unique (peu de langages offrent cette possibilité).

Attention, la fonction « eval » est une technique puissante, mais à utiliser avec parcimonie, car pour certains usages cette technique peut se révéler peu performante, voire entraîner des problèmes de sécurité. C'est la raison pour laquelle vous verrez quelquefois sur internet des articles du genre « eval is evil » (« eval » est le diable). Mais l'utilisation que nous en faisons ici est très pertinente, nous ne prenons aucun risque.

Avant de livrer à votre étude attentive le sketch qui suit, je vais vous fournir quelques précisions :

Le sketch tourne en boucle sur 5 types d'affichage différents :

1 = tracé de points (de 3 pixels de large)

2 = tracé de lignes liant les points N et N-1, chaque point N étant également lié au centre du canvas par une ligne

3 = idem 1, mais les points sont ici remplacés par des sphères transparentes

4 = combinaison des modes 2 et 3

5 = idem 4, avec en plus la possibilité de tracer soi-même avec la souris des lignes reliées au centre du canvas

La courbe polaire a été précalculée une seule fois, ensuite elle est réutilisée pour chacun des affichages que je viens de citer. Les coordonnées de la boucle sont les mêmes, c'est juste le rendu qui change.

```
"use strict";

var sx = 800;
var sy = 600;

var hx, hy;
var points = [];
var max_points = 0;
var i_points = 0;

var fnaContainer = [];
fnaContainer.push('z * z * sin(1/z)');
fnaContainer.push('1');
fnaContainer.push('sin(z)');
fnaContainer.push('sin(2 * z)');
fnaContainer.push('sin(7 * z)');
fnaContainer.push('1 + 2 * cos(z)');
fnaContainer.push('1 + cos(z)');
fnaContainer.push('1 + sin(2 * z)');
fnaContainer.push('1 + 2 * cos(4*z)');
fnaContainer.push('sin(5 * z)');
fnaContainer.push('sin(6 * z)');
fnaContainer.push('sin(8 * z)');
fnaContainer.push('cos(z) * sin(2*z)');
var fnaMessage = '';
```

```

// Liste des modes d'affichage :
// 1 = tracé de points (de 3 pixels de large)
// 2 = lignes liant point N et N-1, chaque point N est lié au centre de
//      l'écran par une ligne
// 3 = idem 1 mais les points sont remplacés par des sphères transparentes
// 4 = combinaison des modes 2 et 3
// 5 = idem 4 avec en plus la possibilité de tracer avec la souris des
//      lignes reliées au centre du canvas
var mode = 1;

function setup() {
    createCanvas(sx, sy);
    background('grey');
    stroke('black');

    // fonction de calcul prise au hasard dans le tableau "fnaContainer"
    var icontainer = int(random(0, fnaContainer.length-1));
    var fnaActive = fnaContainer[icontainer];
    var fna = eval('fna = function(z) { return ' + fnaActive + ' }');

    // préparation du message contenant la formule de calcul
    fnaMessage = 'fonction => ' + fnaActive;
    var a = int(random(1, 9)); // a = 5 & b = 5 => belle spirale
    var b = int(random(1, 6));
    fnaMessage += ' ; a = ' + a;
    fnaMessage += ' ; b = ' + b;

    hx = sx / 2;
    hy = sy / 2;

    // recherche par approximation de la valeur la plus élevée de "r"
    var z;
    var m = 1.0e-30;
    for (z = 0; z <= TWO_PI; z += .1) {
        r = abs(fna(z));
        if (m < r) m = r + 0.1;
    }

    // boucle de précalcul des points de la courbe polaire
    var r, u, v;
    for (z = 0; z <= TWO_PI; z += .01) {
        r = fna(z);
        u = hx + hy * r * cos(a * z) / m;
        if (u >= 0 && u <= sx) {
            v = hy + hy * r * sin(b * z) / m;
            if (v >= 0 && v <= sy) {
                points.push({u: u, v: v});
            }
        }
    }
    max_points = points.length;
    if (max_points > 0) {
        i_points = 1;
    } else {
        console.log('Calcul KO');
        noLoop();
    }
}

```

```

        }
        smooth();
    }

function draw() {
    // points N-1 et N
    var tmp_point_prec = points[i_points - 1];
    var tmp_point = points[i_points];

    // affichage de la formule de calcul en haut à gauche du canvas
    stroke('white');
    strokeWeight(1);
    textSize(18);
    text(fnaMessage, 110, 30);

    stroke(242, 204, 47, 102);

    if (mode == 1) {
        strokeWeight(3);
        point(tmp_point.u, tmp_point.v);
    } else {
        strokeWeight(1);
    }

    fill(100, 204, 47, 20);

    if (mode == 2 || mode == 4 || mode == 5) {
        line(tmp_point_prec.u, tmp_point_prec.v, tmp_point.u, tmp_point.v);
        line(hx, hy, tmp_point.u, tmp_point.v);
    }

    if (mode == 3 || mode == 4 || mode == 5) {
        ellipse(tmp_point.u, tmp_point.v, 10, 10, 20);
        ellipse(tmp_point.u, tmp_point.v, 20, 20, 10);
    }

    if (mode == 5) {
        line(mouseX, mouseY, tmp_point.u, tmp_point.v);
        ellipse(mouseX, mouseY, 10, 10, 20);
    }

    i_points++;
    if (i_points >= max_points) {
        // sauvegarde de l'image finale avant de passer au mode suivant
        save("czes-" + frameCount + ".png");
        i_points = 1;
        mode += 1;
        background('grey');
        if (mode > 5) {
            // on repasse au mode n°1 (la boucle est bouclée :)
            mode = 1;
        }
    }
}

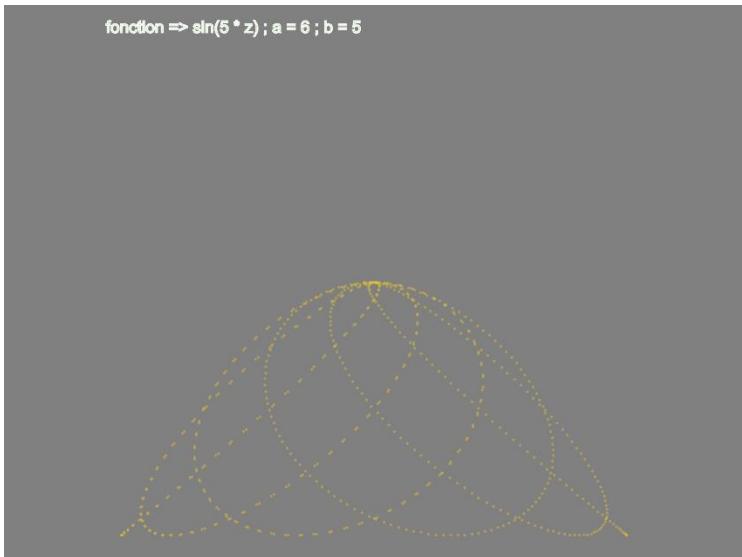
```

```

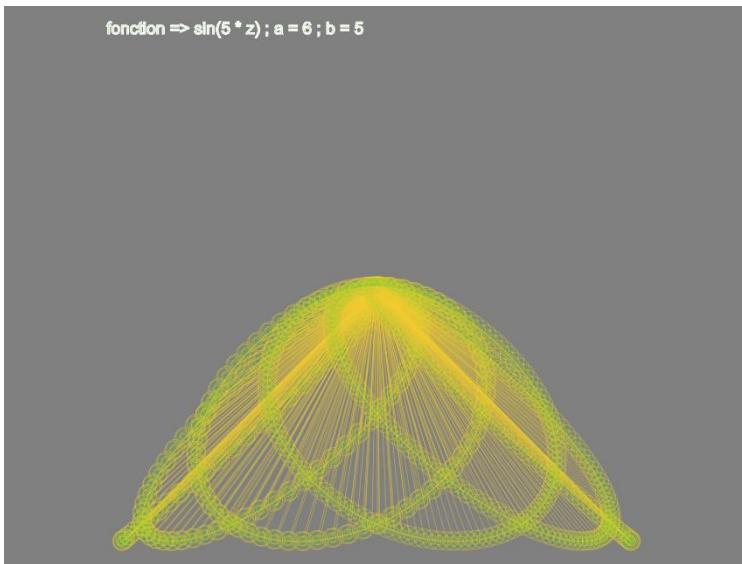
function keyPressed() {
  if (key == 'x' || key == 'X') {
    noLoop();
  } else {
    if (key == 's' || key == 'S') {
      // sauvegarde à la demande
      save("czes-" + frameCount + ".png");
    }
  }
}

```

Exemple de courbe obtenue avec le mode 1 :



Exemple de courbe obtenue avec le mode 4 :



Sur ma page Codepen, ainsi que sur ma page OpenProcessing, vous trouverez une petite variante du sketch précédent. Dans cette variante, le mode d'affichage numéro 3 est occulté, mais surtout après le mode d'affichage numéro 5, on repasse au numéro 1 avec une toute nouvelle formule prise au hasard dans le tableau fnaContainer.

<https://www.openprocessing.org/sketch/410754>

<http://codepen.io/gregja/pen/VpaNOG>

3.2 Curve Stitching

Je me souviens d'une activité, à la fois manuelle et artistique, qui avait le vent en poupe vers la fin des années 70, et même au début des années 80. Elle consistait à dessiner des formes sur une planche de bois, à planter de petites pointes à intervalles réguliers sur les contours de cette forme, et à tendre du fil de couleur (généralement du fil de couture) entre les pointes. On obtenait des formes amusantes, et souvent très intéressantes, d'autant que la lumière jouait sa partition à travers les fils, contribuant à l'étrangeté de l'œuvre ainsi réalisée. Je me souviens d'en avoir réalisé un quand j'étais enfant. C'était amusant à faire mais il fallait une bonne dose de patience pour fixer les fils – et surtout les tendre correctement - sur les têtes des pointes. J'ai réitéré l'expérience récemment avec mes deux enfants (qui ont 5 et 9 ans), et ils ont beaucoup aimé. Le plus dur aura été de trouver une planche de bois adaptée, au magasin de bricolage du coin. Car je vous garantis que ni le medium, ni l'aggloméré ne sont adaptés à ce type d'activité (or, on ne trouve plus que ça dans certains magasins, c'est honteux 😞).

Avec l'arrivée des micro-ordinateurs familiaux au début des années 80, cette activité manuelle a trouvé son pendant électronique. Certains s'amusaient à dessiner sur leur écran des formes géométriques avec le langage LOGO, d'autres avec le langage BASIC. Ce domaine était tout nouveau et très stimulant, on trouvait des livres d'initiation à la programmation qui proposaient des algorithmes pour tracer des formes telles que l'astroïde ou l'épicycloïde (sur lesquelles nous allons revenir dans ce chapitre). Je peux vous dire que j'ai à l'époque testé sur mon Commodore 64, à peu près tout ce qui me tombait sous la main.

Ce que je ne savais pas à l'époque, c'est que ces constructions géométriques avaient une histoire, car elles étaient le fruit de travaux de recherche menés à la fois en France et en Angleterre, pour une grande part à la fin du 19^{ème} siècle, et même avant dans le cas de certaines formes graphiques comme la « courbe de poursuite » dont nous reparlerons dans un instant.

Il faut dire que certains concepts graphiques, comme la « courbe de poursuite » trouvait des applications pratiques dans le domaine de la balistique. Elles intéressaient donc les militaires, et bien sûr les scientifiques.

Mais ces formes graphiques présentaient aussi un fort intérêt pédagogique, pour l'enseignement de la géométrie aux enfants. Mary Everest Boole (1832 – 1916) était une brillante mathématicienne autodidacte, anticonformiste et un peu féministe sur les bords. Elle était l'épouse d'un non moins illustre mathématicien (et également autodidacte), j'ai nommé George Boole. Ce même George Boole qui sera à l'origine de

l'algèbre booléenne, que vous utilisez sans le savoir quand vous écrivez des tests (if) dans vos sketchs P5.

Passionnée par l'enseignement des mathématiques, et des sciences en général, Mary Everest Boole mena des travaux précurseurs dans ce domaine, travaux qui n'ont probablement jamais été diffusés en France et qui semblent être tombés dans l'oubli aujourd'hui. La logique à mettre en œuvre pour réaliser certaines formes géométriques, amener les enfants à penser ces formes « en mouvement », plutôt que de manière statique, c'était là quelques uns de ses axes de recherche. Une de ses amies, Edith L. Somerwell, publia en 1906 un petit livre mettant en lumière le travail pédagogique de Mary Everest Boole. Ce livre s'intitule « A rhythmic Approach To Mathematics », on parvient encore à le trouver sur des sites de vente de produits d'occasion. Ce petit livre est très intéressant, à la fois d'un point de vue pédagogique et historique.

Ces techniques que Mary Everest Boole s'efforçait de promouvoir dans l'enseignement ont pris le nom de « string art » (l'art de la corde) ou encore de « curve stitching » (que l'on pourrait peut-être traduire par « courbes en piqûres »). Si vous recherchez ces termes sur Youtube, vous verrez que ces techniques connaissent encore aujourd'hui un grand succès, un peu partout dans le monde.

En 1989, John Millington publia un très beau livre, dont je n'ai découvert l'existence que très récemment :

« Curve Stitching: Art of Sewing Beautiful Mathematical Patterns », tarquingroup.com (réédité en 1996, malheureusement épuisé depuis, on arrive à le trouver d'occasion)

Cet ouvrage, qui à ma connaissance n'a jamais été traduit en français, constitue une très belle synthèse de l'état de l'art en matière de « curve stitching ». John Millington a conçu son livre en 3 parties :

- une première partie expliquant comment réaliser manuellement quelques une des figures les plus emblématiques de ce courant artistique
- une seconde partie présentant une trentaine de programmes écrits en BASIC, pour dessiner ces formes sur un écran d'ordinateur
- une troisième partie consacrée à l'étude théorique et mathématique, de ces formes géométriques

Je vais dans ce chapitre vous présenter quelques formes géométriques, et nous allons voir comment les représenter graphiquement avec P5.

3.2.1 Courbe de poursuite (« curve of pursuit »)

Drôle de nom, mais vous allez tout de suite comprendre. Imaginez un triangle avec 3 extrémités A, B et C. Sur le coin A vous trouvez un lapin, sur le coin B se trouve un renard qui n'a qu'une idée en tête... vous devinez ?

Joyeux et bondissant, le lapin part en ligne droite vers le coin C.

Le renard part « ventre à terre » en direction du coin A, bien décidé à « boulotter » le malheureux lapin.

Le renard relève la tête de temps à autre, et s'aperçoit qu'il doit réviser sa trajectoire régulièrement, sinon il ratera sa cible. C'est là que se dessine la fameuse « courbe de poursuite ».

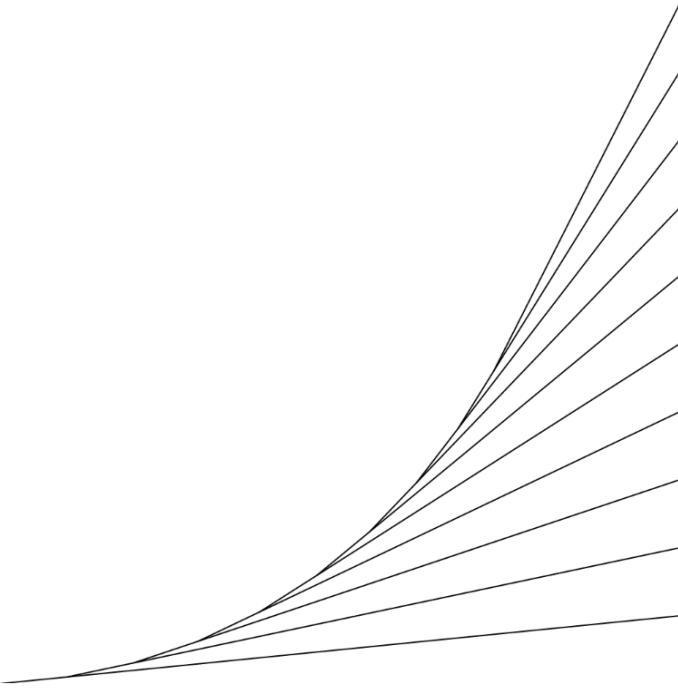
Vous pourriez vous dire que le renard est idiot, et qu'il ferait mieux de se diriger directement vers le coin C, sauf que le renard ne connaît pas la destination du lapin, et s'il anticipe trop, et que le lapin change d'avis entre temps, le renard risque de rater sa cible, et de rentrer le ventre vide, la queue entre les jambes, comme on dit...

Bon, mais alors, ça s'écrit comment une « courbe de poursuite » ? Voici une adaptation pour P5 du programme BASIC proposé par John Millington :

```
function setup() {
  createCanvas(600, 500);
  var x = width;
  var y = height;
  var a = 50;
  var d = y;
  var l = 50;
  for (var n = y ; n >= 0 ; n -= l) {
    line(a, d, x, n);
    var h = sqrt((x-a)*(x-a) + (n-d)*(n-d));
    a = a + l * (x - a) / h;
    d = d + l * (n - d) / h;
  }
  line(x, n, x, y);
  noLoop();
}

function draw() {
```

}



On voit ici chacune des trajectoires que le renard a empruntées, avant de bifurquer pour tenter de se rapprocher du lapin. Il est amusant de noter que, si les deux animaux courent à la même vitesse, le renard ne parviendra jamais à rattraper le lapin.

Bon, notre sketch fonctionne, mais cela manque de vie et on ne sait pas très bien ce qui se passe dans cette « courbe de poursuite ».

Voici une autre version, dans laquelle nous avons précalculé et stocké un tableau de coordonnées, et exploité la fonction « draw » pour mettre nos données en mouvement. C'est la même technique que nous avions employée lors de notre étude sur les cercles et sur les courbes polaires.

```
var datas = [];
var data_i, data_max;
function setup() {
    createCanvas(600, 500);
    var x = width;
    var y = height;
    var a = 150;
    var d = y;
    var l = 50;
    for (var n = y ; n >= 0 ; n -= l) {
        //line(a, d, x, n);
        datas.push({x1:a, y1:d, x2:x, y2:n});
        var h = sqrt((x-a)*(x-a) + (n-d)*(n-d));
        a = a + l * (x - a) / h;
        d = d + l * (n - d) / h;
    }
}
```

```

//line(x, n, x, y);
datas.push({x1:x, y1:n, x2:x, y2:y});
data_i = 0;
data_max = datas.length;
if (data_max <= data_i) {
    noLoop();
}
frameRate(10);
}

function draw() {
    line(datas[data_i].x1, datas[data_i].y1,
        datas[data_i].x2, datas[data_i].y2);
    data_i++;
    if (data_max <= data_i) {
        noLoop();
    }
}

```

Dans son exemple de programme BASIC, John Millington permettait à l'utilisateur de modifier dynamiquement la valeur de la variable « I », que j'ai fixée à « 50 » dans l'exemple de sketch ci-dessus. Vous vous souvenez sans doute que nous avons utilisé précédemment la fonction « prompt » pour mettre en place un formulaire de saisie simplifié. Le code de gestion d'une saisie ressemblait à ceci :

```

var x1 = NaN;
while (Number.isNaN(x1)) {
    x1 = parseInt(prompt('coordonnée X du point n°1 ? ', '10'));
}

```

Je vous propose de reprendre le principe mais de l'améliorer en créant une fonction que nous pourrons réutiliser plus facilement par la suite. Voici le code la fonction :

```

/**
 * Fonction saisie_utilisateur
 * @param {String} message
 * @param {Integer} valeur_defaut
 * @returns {Number|*}
 */
function saisie_utilisateur (message, valeur_defaut) {
    saisie = NaN;
    while (Number.isNaN(saisie)) {
        saisie = parseInt(prompt(message, valeur_defaut));
    }
    return saisie;
}

```

On voit que la fonction « saisie_utilisateur » reçoit deux paramètres en entrée, le message et la valeur par défaut. Nous pourrons ainsi personnaliser message et valeur par défaut à chaque appel de notre fonction personnalisée. Les deux paramètres sont transmis tels quels à la fonction « prompt ». L'instruction « return saisie » en fin de

fonction est très importante, si on l'oublie, la fonction ne renverra aucune valeur et sera inutilisable. C'est une erreur fréquente que d'oublier de placer un « return » à la fin d'une fonction. Même les développeurs professionnels se font avoir de temps en temps.

Vous noterez également la manière dont j'ai défini le bloc de commentaire qui se trouve juste avant la fonction. Ce bloc démarre avec « /** », ce qui signifie qu'il s'agit d'une documentation destinée à être prise en charge par certains logiciels de gestion de documentation utilisé sur les projets professionnels. Cela permet également, avec certains outils de développement (comme Netbeans, Eclipse, PHPStorm, etc..) de profiter d'une auto-complétion améliorée, comme dans l'exemple suivant :

```
var l = saisie_utilis
var l saisie_utilisateur ([String] message, [Integer] valeur_defaut) (titi.js, 365p5/tuto)
```

Dans l'exemple ci-dessus, le logiciel PHPStorm détecte que j'ai saisi « saisie_utilis » et reconnaît qu'une fonction commençant par cette chaîne de caractères existe dans l'application. Du coup il me propose cette fonction en « auto-complétion » en m'indiquant les paramètres attendus par la fonction, ainsi que leurs types. C'est très pratique, quand on y a pris goût, on ne peut plus s'en passer 😊.

Il nous reste à placer l'appel de la fonction à l'intérieur de notre sketch :

- on va donc remplacer la ligne suivante :

```
var l = 9;
```

- par la ligne suivante :

```
var l = saisie_utilisateur ("Longueur d'une étape ? (1 à 30 recommandés)", 9);
```

La « courbe de poursuite » est un sujet qui a fait couler beaucoup d'encre, et sans doute mis en surtension pas mal de neurones. Si vous avez envie de creuser le sujet, voici quelques pistes de lecture :

<http://mathworld.wolfram.com/PursuitCurve.html>

<http://sections.maa.org/okar/papers/2006/lloyd.pdf>

https://en.wikipedia.org/wiki/Pursuit_curve

3.2.2 Tractrix

Tractrix reprend en partie le principe de la « courbe de poursuite », mais il y a une différence de fond, à savoir que le poursuivant ne peut jamais rattraper la cible, la distance entre les deux demeurant constante en toutes circonstances.

Si l'on oublie le lapin et le renard, disons que nous avons 2 points reliés par un joint rigide, il y a point guideur (ou tracteur) et un point suiveur. Tractrix, c'est en quelque sorte l'analyse du mouvement du point suiveur.

Peut être avez-vous déjà vu une voiture en panne se faire tracter par une autre via une barre rigide fixée entre les 2 véhicules. Le véhicule suiveur peut légèrement osciller à l'arrière de la trajectoire du véhicule tracteur, mais sa marge de manœuvre est très réduite et il ne peut jamais se rapprocher du tracteur, étant donné que la barre est rigide. On peut aussi imaginer le cas de figure d'une personne tenant une barre rigide dans une main par une extrémité, et laissant l'autre extrémité trainer derrière elle, creusant ainsi un sillon dans le sol. C'est ce sillon que l'algorithme connu sous le nom de Tractrix va nous permettre de dessiner.

```
/*
 * Fonction saisie_utilisateur
 * @param {String} message
 * @param {Integer} valeur_defaut
 * @returns {Number|*}
 */
function saisie_utilisateur (message, valeur_defaut) {
    saisie = NaN;
    while (Number.isNaN(saisie)) {
        saisie = parseInt(prompt(message, valeur_defaut));
    }
    return saisie;
}

var datas = [];
var data_i, data_max;
function setup() {
    createCanvas(610, 500);
    var x = width-10;
    var y = height-20;
    var a = 0;
    var d = y;
    var c = 50;
    var k = x - c;
    var l = saisie_utilisateur("Longueur d'une étape? (1 à 30 recommandés)",
        9);
    var steps = 100;
    var pas = 1;
```

```

for (var n = 1, nmax = steps/l+1 ; n <= nmax ; n += pas) {
  var h = sqrt(2 * k * a - a * a);
  datas.push({x1:a+c, y1:d, x2:x, y2:d-h});
  a = a + l * (k - a) / k;
  d = d - l * h / k;
}

data_i = 0;
data_max = datas.length;
console.log(data_max);
if (data_max <= data_i) {
  noLoop();
}
frameRate(10); // ralentir l'animation à 10 fps

// légende
textSize(14);
ellipse(25, 30, 3);
stroke('blue');
text("suiveur", 30, 30);
stroke('red');
ellipse(105, 30, 3);
text("tracteur", 110, 30);

}

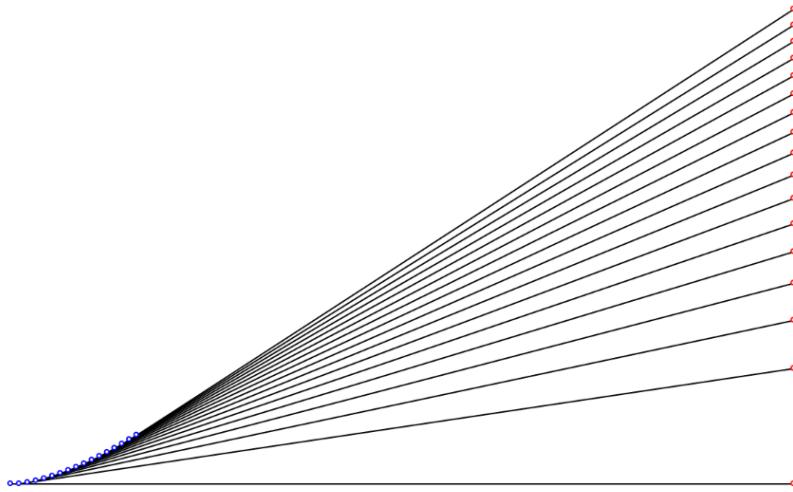
function draw() {
  stroke('black');

  line(datas[data_i].x1, datas[data_i].y1,
        datas[data_i].x2, datas[data_i].y2);
  stroke('blue');
  ellipse(datas[data_i].x1, datas[data_i].y1, 3);
  stroke('red');
  ellipse(datas[data_i].x2, datas[data_i].y2, 3);
  data_i++;
  if (data_max <= data_i) {
    noLoop();
  }
}

```

Voici ce que vous devriez obtenir pour une valeur de « l » égale à 6 :

• suiveur • tracteur



L'algorithme en lui-même n'est pas franchement passionnant, mais le principe de cette forme géométrique me semble intéressant à connaître.

On peut noter que Tractrix intéresse beaucoup les ingénieurs et les chercheurs, notamment dans les domaines de la mécanique et de la robotique. Si vous avez envie d'approfondir le sujet, voici quelques pistes de lecture :

<https://en.wikipedia.org/wiki/Tractrix>

<http://www.mecheng.iisc.ernet.in/~asitava/jmr-1.pdf>

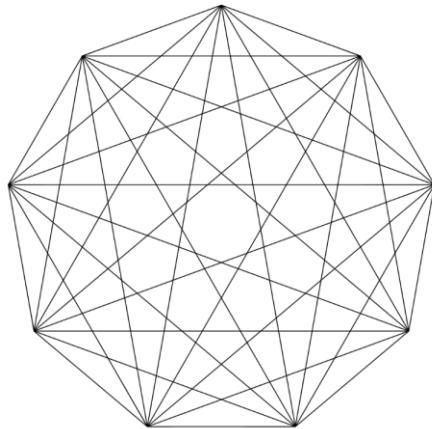
<http://www.mecheng.iisc.ernet.in/~asitava/mmt-hyper.pdf>

<https://ir.library.oregonstate.edu/xmlui/bitstream/handle/1957/26312/Ekert.pdf>

3.2.3 Rose mystique

Je connaissais cette construction géométrique sous le nom de « napperon », car c'était le nom que lui avait donné l'auteur d'un de mes premiers livres de programmation. J'ai découvert récemment que nos amis anglophones lui donnent le nom de « mystic rose ».

Voici l'exemple de la rose mystique à 9 points :

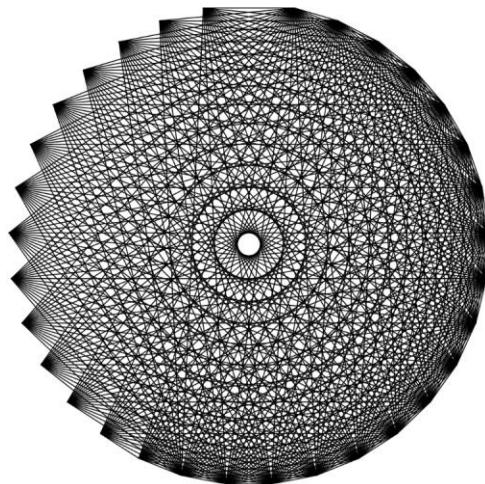


Le principe consiste ici à découper notre cercle en 9 points et à joindre chacun de ces 9 points à l'ensemble des autres points.

Vous remarquerez un détail amusant, la forme au centre a 9 côtés, exactement comme la forme extérieure. Faites un essai avec d'autres valeurs impaires, vous retrouverez le même phénomène. Observez aussi ce qui se passe pour les valeurs 1, 2, 3, 4.

Pour des valeurs élevées, l'effet obtenu fait vraiment penser aux napperons que brodaient les grands-mères... euh, il y a longtemps... très très longtemps (il est vrai que tout se perd).

J'ai fait une copie d'écran de la forme ci-dessous alors qu'elle était encore en construction. Sympa, non ?



Voici le sketch du napperon, pardon... de la rose mystique :

```
function saisie_utilisateur (message, valeur_defaut) {
    saisie = NaN;
    while (Number.isNaN(saisie)) {
        saisie = parseInt(prompt(message, valeur_defaut));
    }
    return saisie;
}
var datas = [];
var data_i, data_max;
function setup() {
    createCanvas(610, 500);
    var x = width/2;
    var y = height/2;
    var v = saisie_utilisateur ("Combien de points ?", 9);
    var p = TWO_PI/v;
    for (var b = p ; b <= TWO_PI; b += p) {
        for (var h = b + p ; h <= TWO_PI ; h += p) {
            datas.push({
                x1:x+y*sin(b), y1:y-y*cos(b),
                x2:x+y*sin(h), y2:y-y*cos(h)
            });
        }
    }
    data_i = 0;
    data_max = datas.length;
    console.log(data_max);
    if (data_max <= data_i) {
        noLoop();
    }
    frameRate(10); // ralentir l'animation à 10 fps
}

function draw() {
    line(datas[data_i].x1, datas[data_i].y1,
          datas[data_i].x2, datas[data_i].y2
    );
    data_i++;
    if (data_max <= data_i) {
        noLoop();
    }
}
```

La rose mystique est une construction assez fascinante, et j'ai remarqué qu'elle plaît beaucoup aux enfants. Ceci dit, il y a une construction qui amuse encore plus les enfants – vous allez comprendre pourquoi dans un instant – c'est l'épicycloïde.

3.2.4 Epicycloïde

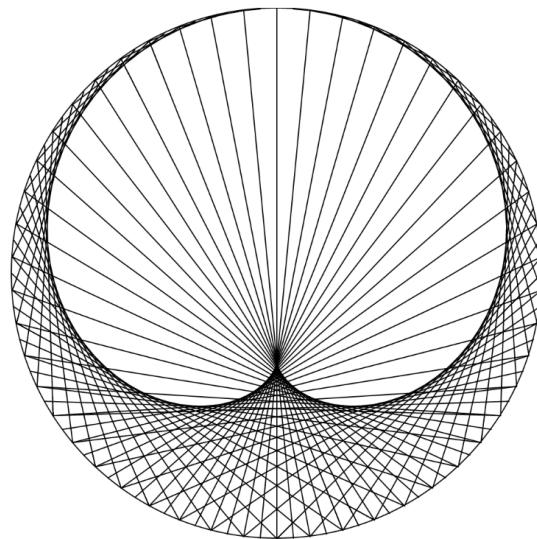
Le terme épicycloïde englobe une famille de formes très différentes et extrêmement intéressantes à observer.

Avant de lancer le calcul permettant de générer un épicycloïde, on va demander à l'utilisateur de saisir un nombre de « cusps ». Pardon, « cusps », c'est le terme anglais. En français, on parlerait de « cuspides ». What ! C'est quoi cette bête ? J'ai regardé dans un dico, voici une définition que je trouve assez parlante :

« *Une extrémité pointue où deux courbes se rejoignent* »

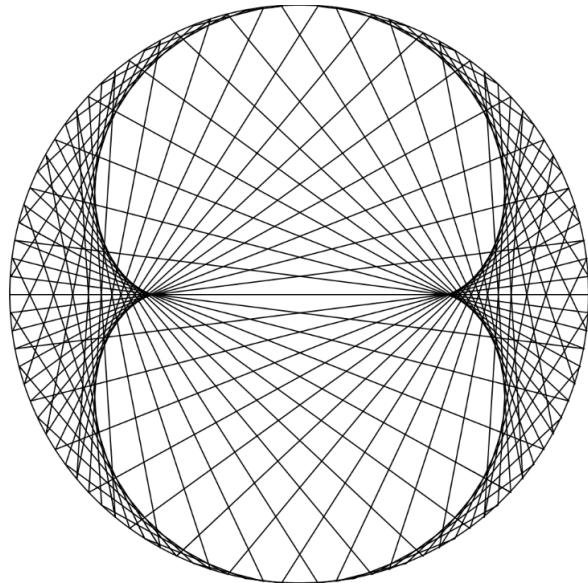
Un synonyme de cuspide ? Hmm... « rebroussement », ça vous va ? Les espagnols parlent de « cuspide » au sens de « rebroussement » et les italiens le comprennent au sens de « point de régression » (si j'en crois les quelques recherches rapides que j'ai effectuées sur le sujet).

Vous ne comprenez probablement toujours pas ce qu'est un cuspide. OK, je lève le voile... TATAM !! voici un exemple d'épicycloïde à 1 rebroussement (ou 1 cuspide) :



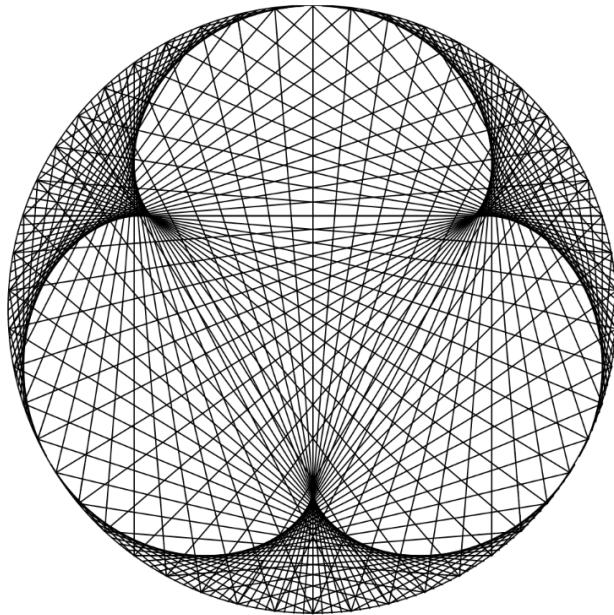
Ce type d'épicycloïde porte le joli nom de « cardioïde » car il rappelle vaguement la forme d'un cœur. Si vous montrez cette forme à des enfants, il y a de grandes chances pour qu'ils vous disent que ça ressemble plutôt à une (belle) paire de fesses... La vérité sort toujours de la bouche des enfants, enfin c'est ce qu'on dit...

Avec 2 rebroussements, on obtient un « néphroïde » :



Ah oui, les deux formes que l'on vient de voir ont été tracées en demandant au sketch de placer 100 points à l'intérieur du cercle principal (et bien sûr de les lier).

Avec 3 rebroussements, on obtient l'épicycloïde de Crémone, du nom d'un célèbre mathématicien. Voici le résultat obtenu pour 200 points :



Je vous invite à tester le sketch qui suit avec toutes les combinaisons de valeurs qui vous passent par la tête.

```
function saisie_utilisateur (message, valeur_defaut) {
    saisie = NaN;
    while (Number.isNaN(saisie)) {
        saisie = parseInt(prompt(message, valeur_defaut));
    }
    return saisie;
}

var datas = [];
var data_i, data_max;

function setup() {
    createCanvas(610, 500);
    var x = (width-10)/2;
    var y = (height-20)/2;
    var v = saisie_utilisateur ("Combien de rebroussements ?", 1);
    var n = saisie_utilisateur ("Combien de points sur le cercle ?", 100);
    ellipse(x, y, y*2);
    var p = TWO_PI / n;
    for (var b = 0 ; b <= TWO_PI; b += p) {
        var h = b * (v + 1);
        datas.push({
            x1:x+y*sin(b), y1:y-y*cos(b),
            x2:x+y*sin(h), y2:y-y*cos(h)
        });
    }
    data_i = 0;
    data_max = datas.length;
    if (data_max <= data_i) {
        noLoop();
    }
    frameRate(10); // ralentir l'animation à 10 fps
}

function draw() {
    line(datas[data_i].x1, datas[data_i].y1,
          datas[data_i].x2, datas[data_i].y2);

    data_i++;
    if (data_max <= data_i) {
        noLoop();
    }
}
```

Si vous avez envie d'approfondir ce sujet, je vous recommande vivement les deux vidéos Youtube ci-dessous, une de la chaîne MicMaths et l'autre de la chaîne Mathologer. Vous y trouverez deux lectures différentes du même sujet. C'est juste passionnant 😊 :

- [La face cachée des tables de multiplication](#) de Micmaths (Mickaël Launay)
- [Times Tables, Mandelbrot and the Heart of Mathematics](#) de Mathologer

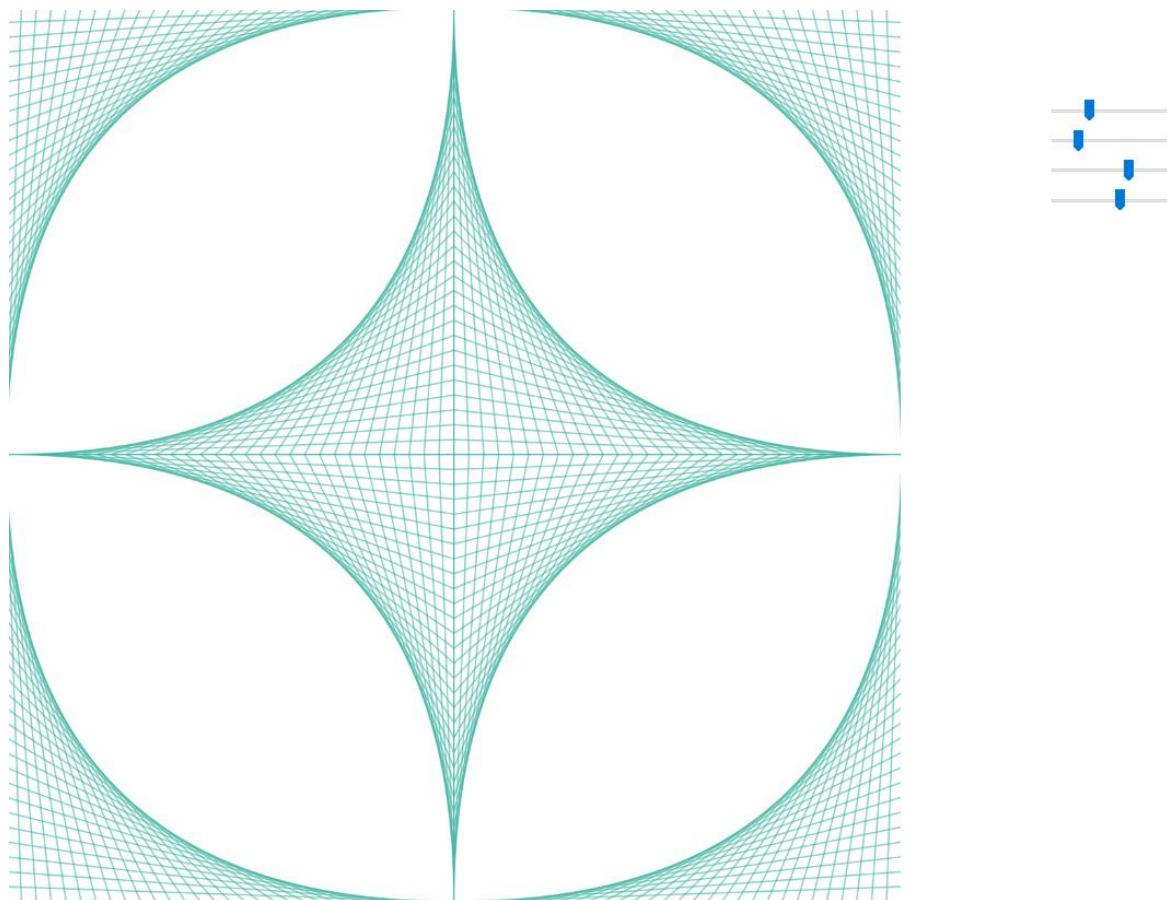
3.2.5 Huit paraboles dans un carré, avec P5.dom

Dans les exemples proposés par John Millington dans son livre, il y en a qui s'appelle « 8 paraboles dans un carré ». La forme obtenue est intéressante, mais j'ai surtout envie de m'amuser avec la librairie – complémentaire à P5 – qui s'appelle P5.dom. Cette librairie permet d'interagir facilement avec le navigateur en ajoutant des champs de saisie, des boutons, et des sliders (qui sont des sortes de curseurs). Ce sont ces sliders que je vais utiliser ici pour modifier le comportement du graphique en cours d'animation.

Le premier slider va me permettre d'accélérer ou de ralentir l'animation.

Les 3 sliders suivants vont me permettre de doser dynamiquement les niveaux de rouge, de vert et de bleu de l'image en train de se générer.

A l'écran, ça ressemble à ça :



J'ai également ajouté à mon sketch les règles de fonctionnement suivantes :

Touches associées au générateur de graphe

- X = Arrêt/Redémarrage de l'animation
 - S = Sauvegarde du graphe en cours
 - G = Bascule entre Affichage 1 (*) et Affichage 2 (*)
 - R = Rotation (sur Affichage 1 seulement)
 - ... plus un slider pour jouer sur le nombre de FPS, et 3 sliders pour les couleurs (RVB)

(*)

- l'affichage 1 simule une sorte de morphing en partant d'un graphe à un seul segment, pour aller progressivement vers un graphe composé de 40 segments. On repart ensuite dans un cycle inverse (de 40 à 1), avant de repartir dans un nouveau cycle.
 - Dans le tracé progressif de l'affichage 2, on stoppe l'incrémentation du nombre de segments, et on redessine la forme en cours, segment par segment, ce qui permet de mieux comprendre comment fonctionne l'algorithme effectuant le tracé. A la fin de l'animation, l'image se fige, ce qui laisse de la sauvegarder si on le souhaite (via la touche S) et on peut relancer l'animation en pressant 2 fois la touche X.

Dans les coulisses, le code source du sketch ressemble à ceci :

```
var v=1 ;
var inc=1;
var record = false;
var datas = [];
var rotation = false;
var anglerot = 0;
var frameslider;
var redslider;
var grnslider;
var bluslider;
var mystop = false;

function setup() {
  createCanvas(600,600);
  background(255);
  smooth();
  frameslider = createSlider(10, 60, 25, 1);
  frameslider.position(700, 620);
  frameslider.style('width', '80px');
  redslider = createSlider(0, 255, 0, 1);
  redslider.position(700, 640);
  redslider.style('width', '80px');
  grnslider = createSlider(0, 255, 0, 1);
  grnslider.position(700, 660);
  grnslider.style('width', '80px');
  bluslider = createSlider(0, 255, 0, 1);
  bluslider.position(700, 680);
  bluslider.style('width', '80px');
}

function draw() {
  frameRate(frameslider.value());
}
```

```

stroke(redslider.value(), grnslider.value(), bluslider.value(), 150);

if (!record && !rotation) {
    background(255);
}
if (v>40) {
    inc=-1;
}
if (v<2) {
    inc=1;
}
if (rotation && !record) {
    translate(width/2, height/2);
    anglerot +=1;
    if (anglerot > 360) {
        anglerot = 0;
    }
    rotate(anglerot);
}
if (!record) {
    if (!rotation) {
        v+=inc;
    }
    graph(false);
} else {
    if (iter >= datas.length-1) {
        noLoop();
        mystop = true;
    }
    if (iter >= datas.length) {
        iter = 0;
    }
    if (iter == 0) {
        background(255);
    }
    line(datas[iter].x1, datas[iter].y1, datas[iter].x2, datas[iter].y2) ;
    iter +=1;
}
//if (frameCount>500) noLoop();
}

function graph(recdatas) {
    if (recdatas) {
        datas = [];
        iter = 0;
    }
    var r, g, t, h=0;
    var x=width/2;
    var y=height/2;
    var x1, y1, x2, y2;
    for (h=0; h<=y+1; h+=y/v) {
        for (t=-1; t<=1; t+=2) {
            for (g=-1; g<=1; g+=2) {
                for (r=0; r<=1; r+=1) {
                    x1 = x+h*t;
                    y1 = y*(1+g*r);
                    x2 = x+y*t*r;
                    y2 = y*(1+g)-h*g;
                    if (recdatas) {
                        datas.push({x1: x1, y1:y1, x2:x2, y2:y2});
                    } else {
                        line(x1, y1, x2, y2) ;
                    }
                }
            }
        }
    }
}

```

```

        }
    }
}

function keyPressed () {
    if (key=='s' || key == 'S'){
        saveCanvas("stitch01-"+frameCount+".png", 'png');
    }
    if (key=='x' || key == 'X'){
        if (mystop) {
            mystop = false ;
            println('relance');
            loop();
            redraw();
        } else {
            println('nombre de segments lors du stop : ' + v);
            mystop = true ;
            noLoop();
        }
    }
    if (key=='g' || key == 'G'){
        /*
         * On inverse la valeur du booléen "record"
         * Si record est vrai, alors on retrace la dernière forme en
         * l'enregistrant dans le tableau "
        */
        record = !record;
        if (record) {
            println('tracé progressif pour '+v+' segments');
            graph(true);
        } else {
            println('retour au tracé global (un graphique complet par frame)');
            if (mystop) {
                mystop = false;
                loop();
                redraw();
            }
        }
    }
    if (key=='r' || key == 'R'){
        /*
         * l'activation de la rotation n'est possible que dans le premier
         * mode d'affichage (tracé global à raison d'un graphique par frame)
        */
        if (!record) {
            rotation = !rotation;
        }
    }
}

```

Vous pouvez voir ce sketch en fonctionnement sur Codepen.io

<https://codepen.io/gregja/pen/qreXGQ/>

Pour que le sketch fonctionne dans codepen, il faut le paramétrer de manière à intégrer P5.dom dans la liste des librairies du Pen. Le lien est le suivant :

<https://cdn.rawgit.com/lmccart/p5.js/master/lib addons/p5.dom.js>

Pen Settings

HTML CSS **JavaScript** Behavior

JavaScript Preprocessor

None

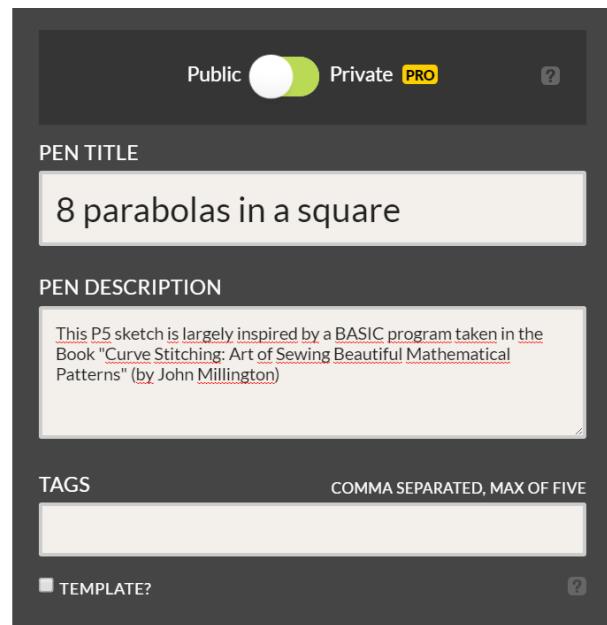
Add External JavaScript

These scripts will run in this order and before the code in the JavaScript editor. You can also link to another Pen here, and it will run the JavaScript from it. Also try typing the name of any popular library.

☰ https://cdnjs.cloudflare.com/ajax/libs/p5.js/0.5.8/p5.min.js
☰ https://cdn.rawgit.com/lmccart/p5.js/master/lib addons/p5.dom.js

Quick-add: ---

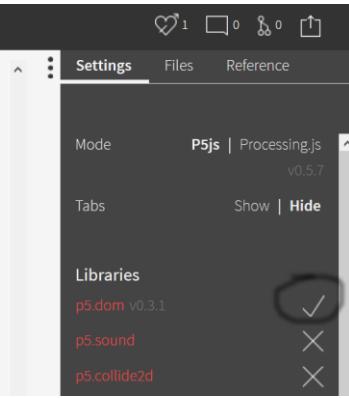
+ add another resource



Le même sketch dans Openprocessing :

<https://www.openprocessing.org/sketch/419827>

Pour que P5.dom fonctionne dans Openprocessing, il faut cocher l'option correspondante dans la liste des librairies associées au sketch :



The screenshot shows the Openprocessing sketch editor interface. On the left is the code editor with the following pseudocode:

```
124     record = !record;
125     if (record) {
126         println('tracé progressif pour '+v+' segments');
127         graph(true);
128     } else {
129         println('retour au tracé global (un graphique complet par frame)');
130         if (mystop) {
131             mystop = false;
132             loop();
133             redraw();
134         }
135     }
136 }
137 if (key=='r' || key == 'R'){
138     /*
```

On the right is the 'Settings' panel. Under 'Mode', it says 'P5js | Processing.js v0.5.7'. Under 'Tabs', it says 'Show | Hide'. Under 'Libraries', there is a list:

Library	Version	Status
p5.dom	v0.3.1	✓
p5.sound		✗
p5.collide2d		✗

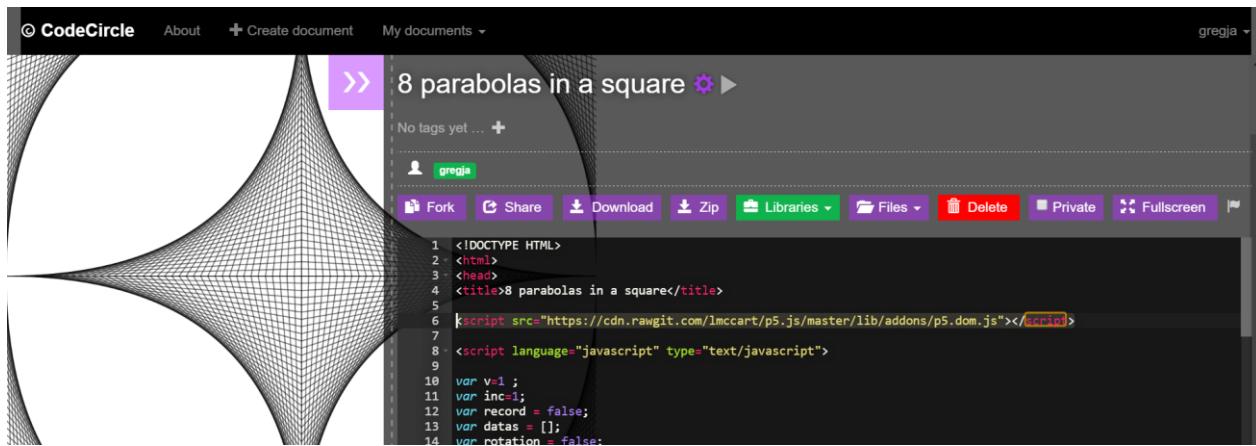
Il y a un petit souci dans Openprocessing, car le Canvas est centré d'office, masquant du coup les 4 sliders dont j'avais défini les positions de manière fixe. Il faudra que je réétudie la question 😊.

Le même sketch avec Live.codecircle.com :

<https://live.codecircle.com/d/jXeswrqW7Jfga4jJv>

Pour que P5.dom fonctionne, il faut l'insérer avec une balise « script », en y plaçant l'attribut « src » avec la même URL que nous avions utilisée dans Codepen :

<https://cdn.rawgit.com/lmccart/p5.js/master/lib addons/p5.dom.js>



Peut être vous êtes-vous demandé d'où je sortais l'URL me permettant de charger P5.dom dans Codepen et dans Codecircle, eh bien je vous avoue que j'ai un peu transpiré avant de trouver la solution.

Car la librairie P5.dom n'est pas référencée sur CDNJS, contrairement à P5. Du coup, j'essayais de charger P5.dom dans Codepen et dans Codecircle à partir de l'URL officielle du projet :

<https://raw.githubusercontent.com/lmccart/p5.js/master/lib addons/p5.dom.js>

... mais cela ne fonctionnait pas, car les navigateurs bloquent ce type d'URL qui ne provient pas d'un tiers de confiance comme CDNJS.

Pour contourner la difficulté, je suis passé par le site suivant :

<http://rawgit.com/>

Je lui ai donné en entrée l'URL de P5.dom, et rawgit.com m'a回报了一条 URL sécurisée que j'ai pu utiliser sur Codepen et sur Codecircle, et qui est la suivante :

<https://cdn.rawgit.com/lmccart/p5.js/master/lib addons/p5.dom.js>

Si vous êtes amené à utiliser d'autres librairies non référencées sur CDNJS, vous pourrez vous en sortir de cette manière.

4. Conclusion

J'aurais souhaité aborder plus de sujets dans ce tutoriel, mais j'ai dû faire des choix, faute de temps.

Parmi les sujets que je souhaiterais aborder dans une prochaine version, il y a notamment :

- Comment créer ses propres objets Javascript, et dans quel cas c'est utile, voire indispensable
- Les méthodes P5 relatives à la translation, la rotation et la mise à l'échelle
- L'utilisation de P5 comme outil de simulation
- Et bien d'autres choses encore...

P5 constitue un formidable terrain d'expérimentation, et un très bel outil pour s'initier à la programmation et au creative coding.

J'ai pris énormément de plaisir à rédiger ce support, j'espère que vous prendrez autant de plaisir en le lisant. J'espère surtout qu'il vous sera utile durant votre apprentissage de P5, qu'il vous apportera suffisamment de « clés » pour que vous puissiez prendre votre envol dans l'univers fabuleux du Creative Coding.

Bon vol, petit bourdon ☺

5. Références bibliographiques

Voici quelques conseils de lecture pour approfondir vos connaissances.

Site officiel P5.js :

<http://p5js.org/>

Un bon tutoriel en français sur P5 :

https://b2renger.github.io/Introduction_p5js/

Un autre bon tutoriel, également en français :

<http://www.lyceelecorbusier.eu/p5js/>

Slide d'introduction à l'animation avec P5.js (avec le code source sur Github), que j'avais créé pour le NUMA (en décembre 2016), et qui a servi de base à certains chapitres de ce support :

<http://backstages.gregphplab.com/circlecirclemeetup/>

<https://github.com/gregja/circleCircleMeetup>

Un dossier sur la problématique de migration de sketchs Processing vers P5 (présenté dans le cadre du meetup CreativeCodeParis en janvier 2017) :

<https://github.com/gregja/p5Migration>

Pour les nostalgiques, et les amateurs de graphiques vintage, ce site est une référence :

<http://recodeproject.com/>

Un site de référence pour les passionnés de géométrie :

<http://mathcurve.com/>

Un excellent tutoriel (en 3 parties et en anglais) consacré à Processing, avec plein de bonnes idées à adapter à P5 :

<https://www.ibm.com/developerworks/library/os-datas/>

(On notera que le site « IBM Developerworks » contient de nombreux articles de qualité consacrés au développement web et au langage Javascript. L'accès est gratuit, les articles librement téléchargeables une fois que vous avez créé un compte).

Lien vers un très bon livre d'introduction à P5 (en anglais) :

<https://p5js.org/books/>

Beaucoup de très bons livres consacrés à Processing :

<https://processing.org/books/>

Vous pourrez retrouver et télécharger ce support sur Github :

<https://github.com/gregja/JSCorner>

Dans ce même dépôt JSCorner, vous trouverez également un cours relatif au langage Javascript et à la norme HTML5 :

The screenshot shows a GitHub repository page for 'gregja / JSCorner'. The repository has 2 commits, 1 branch, 0 releases, and 1 contributor. The license is MIT. The repository was created by 'gregja' and contains files like 'Cours Javascript_HTML5.pdf', 'LICENSE', 'README.md', and 'Tuto_P5_Premiers_pas.pdf'. All files were committed 11 hours ago.

File	Description	Time
Cours Javascript_HTML5.pdf	Premières versions	11 hours ago
LICENSE	Initial commit	11 hours ago
README.md	Premières versions	11 hours ago
Tuto_P5_Premiers_pas.pdf	Premières versions	11 hours ago

Ce cours est à votre disposition, il aborde de nombreuses notions avancées du langage Javascript, il pourra certainement vous être utile si vous souhaitez approfondir vos connaissances sur ce sujet.

A. Annexe

A.1 Window.isNaN ou Number.isNaN

Nous avons évoqué au chapitre 2.7 la problématique des nombres qui n'en étaient pas vraiment, les fameux « NaN » (Not a Number). Peut être vous interrogez-vous sur ce qu'est un nombre qui n'en est pas vraiment un... c'est bien normal. Je vous propose de saisir dans la console du navigateur les lignes suivantes :

```
> var test = 'xxx20';
< undefined
> var test2 = parseInt(test);
< undefined
> test2
< NaN
> typeof test2
< "number"
>
```

La fonction « parseInt » s'est efforcée de créer une variable de type « number ». Pour ce faire, elle a informé l'interpréteur Javascript du fait que la variable « test2 » allait recevoir un nombre, mais elle a échoué à convertir la variable « test » en quelque chose de numérique, alors elle en informe l'interpréteur Javascript en renvoyant cette fameuse valeur « NaN ».

Il existe une fonction `window.NaN` et une fonction `Number.NaN`, laquelle choisir ? Testons-les toutes les deux :

```
> Number.isInteger(test2)
< false
> window.NaN(test2);
< true
```

On pourrait se dire que tout va bien, les 2 fonctions semblent renvoyer le même résultat... OK, alors allons un peu plus loin dans les tests :

```
> var a = 2 / "foo";
var b = "foo";
console.log(a); // NaN
console.log(b); "foo"
console.log(window.isNaN( a )); // true
console.log(window.isNaN( b )); // true -- Oups, ce résultat est absurde !
NaN
foo
true
true
```

On voit que la fonction « window.isNaN » renvoie un résultat absurde sur la variable « b ». Elle n'est pas fiable et il est préférable d'utiliser la fonction « Number.isNaN » :

```
> var a = 2 / "foo";
var b = "foo";
console.log(a); // NaN
console.log(b); "foo"
console.log(Number.isNaN( a )); // true
console.log(Number.isNaN( b )); // false, là c'est correct car cette variable n'est pas de type "number"
NaN
foo
true
false
```

C'est troublant, n'est ce pas ? Dites-vous que vous venez d'apprendre un truc que même certains développeurs JS aguerris ne savent pas. Je rencontre encore trop souvent cette fonction « window.isNaN » dans des projets en entreprise, ce qui est assez désolant compte tenu qu'elle n'est pas fiable.

Partant de cette constatation, vous comprenez peut être mieux pourquoi il est souhaitable de contrôler la validité de la variable « test2 » (de notre exemple précédent) via un test de ce type :

```
> if (typeof test2 === 'number' && !window.isNaN) { console.log('ok'); } else { console.log('bad'); }
bad
```

Fort de ces explications, je vous invite à reconsidérer la boucle de contrôle de saisie que nous avons utilisée au chapitre 2.7, elle vous semblera peut être plus claire maintenant :

```
x1 = NaN;
while (Number.isNaN(x1)) {
    x1 = parseInt(prompt('coordonnée X du point n°1 ? ', '10'));
}
```

A.2 Polygone animé

Voici une version animée du sketch de tracé de polygone du chapitre 2.8.6.

```
var sx = 600;
var sy = 500;
var backcolor = "white";
var drawcolor = "black";
var xc, yc;
var rayon = 200;
var s = 20; // nombre de côtés du polygone
var data = [];
var i, imax;

function setup() {
    createCanvas(sx, sy);
    background(backcolor);
    stroke(drawcolor);
    xc = sx / 2;
    yc = sy / 2;
    for (var a = 0 ; a <= 360 ; a+= 360/s) {
        var x = xc + rayon * cos(radians(a));
        var y = yc + rayon * sin(radians(a));
        if (a != 0) {
            data.push({x1:px, y1:py, x2:x, y2:y});
            data.push({x1:x, y1:y, x2:xc, y2:yc});
        }
        var px = x;
        var py = y;
    }
    i = 0 ;
    imax = data.length;
    frameRate(10); // vitesse très ralentie pour mieux voir l'animation
}

function draw() {
    if (i < imax) {
        line(data[i].x1, data[i].y1, data[i].x2, data[i].y2);
        i += 1;
    } else {
        noLoop();
    }
}
```

Il y a bien évidemment plusieurs manières d'écrire ce sketch, le vôtre diffèrera probablement de ce que je vous propose ici, il sera peut être même mieux écrit. Vous pensez que je rigole ? Pas du tout, je suis même très sérieux.

A.3 Les tableaux en JS

Créer un tableau en Java, par exemple pour un sketch Processing, est une opération que je trouve particulièrement lourdingue. Voici un exemple de déclaration de tableau en Java, il s'agit ici de déclarer un tableau de 500 postes (ou éléments) contenant des nombres en virgule flottante :

```
float [] a = new float[500];
```

En Javascript, c'est tellement plus cool, non seulement on ne s'embête pas à préciser le nombre de postes (sauf cas très exceptionnel), mais en plus on ne précise pas le type, d'autant qu'en Javascript, un tableau peut contenir des données de type très différents.

Voici donc la déclaration d'un tableau JS :

```
var a = [];
```

On pourrait écrire aussi ceci :

```
var a = new Array();
```

Si on souhaite vraiment créer un tableau contenant 500 postes, on peut quand même le faire en JS en écrivant ceci :

```
var a = new Array(500);
```

On peut initialiser le contenu d'un tableau au moment de sa création, de deux manières différentes :

```
// Un tableau créé au moyen du constructeur Array()  
var a = new Array("produit1", "produit2", "produit3");  
  
// Le même tableau créé de manière littérale  
var a = ["produit1", "produit2", "produit3"];
```

Il semble se dégager un consensus au sein de la communauté des développeurs Javascript, pour utiliser la déclaration littérale, de préférence à la déclaration via le constructeur Array().

Il y a en effet un piège dans l'utilisation du constructeur Array(). Si on ne lui transmet qu'un seul paramètre, de type numérique, alors ce paramètre va avoir pour effet de créer un tableau dont le nombre de postes correspond à la valeur de ce paramètre (le paramètre n'est pas conservé pour alimenter le premier poste du tableau). Ce n'est qu'à partir de 2 paramètres, que le constructeur Array() les considère comme des valeurs devant être stockées dans un tableau (et crée le tableau correspondant). Cela peut être

source de confusion, et donc d'erreur.

Après avoir initialisé notre tableau « a », on peut ajouter de nouveaux éléments à la volée, au moyen de la méthode « push » :

```
a.push("produit4") ;  
a.push("produit5") ;
```

On peut connaître la longueur d'un tableau au moyen de la méthode « length » :

```
console.log(a.length()) ; // renverra la valeur 5
```

On peut écrire une boucle qui parcourt l'ensemble des postes d'un tableau :

```
> for (var i = 0, imax = a.length ; i < imax ; i++) {  
    console.log(a[i]);  
}  
produit1  
produit2  
produit3  
produit4  
produit5  
< undefined
```

VM7385:2
VM7385:2
VM7385:2
VM7385:2
VM7385:2

On peut parcourir le tableau dans l'autre sens :

```
> for (var i = a.length - 1 ; i >= 0 ; i--) {  
    console.log(a[i]);  
}  
produit5  
produit4  
produit3  
produit2  
produit1  
< undefined
```

VM10315:2
VM10315:2
VM10315:2
VM10315:2
VM10315:2

La méthode « foreach » constitue un autre moyen très pratique de parcourir un tableau :

```
> a.forEach(function(element, index) {
    console.log(index + ' ' + element  );
  })
0 produit1 VM18339:2
1 produit2 VM18339:2
2 produit3 VM18339:2
3 produit4 VM18339:2
4 produit5 VM18339:2
< undefined
```

On peut aussi parcourir le tableau avec une boucle « while » :

```
> var i = 0;
var imax = a.length;
while (i < imax) {
  console.log(a[i]);
  i++;
}
produit1 VM31076:4
produit2 VM31076:4
produit3 VM31076:4
produit4 VM31076:4
produit5 VM31076:4
< 4
```

On peut aussi créer des tableaux à 2 dimensions ou plus. Mais il y a de petites différences par rapport à Java. En effet, en Java, on pourrait écrire ceci :

```
float [][] a = new float[500][2];
```

Mais en Javascript, on ne peut initialiser que le premier niveau :

```
var a = new Array(500);
```

Mais rien ne nous empêche d'écrire une petite boucle qui parcourt les 500 postes et pour chaque poste initialise un tableau à deux dimensions :

```
var i, j ;
for (i = 0; i < 500; i++){
  a[i] = new Array(2);
  for (j = 0; j < 2; j++){
    a[i][j] = random(10,490);
  }
}
```

On retrouve ces techniques dans un sketch de Paul Orlov :

Code P5	Code Processing d'origine
<pre> var a = new Array(500) ; function setup() { createCanvas(700, 500); var i, j ; for (i = 0; i < 500; i++){ a[i] = new Array(2); for (j = 0; j < 2; j++){ a[i][j] = random(10,490); } } } function draw() { smooth(); noStroke(); background(0); var i, eDist, eSize, eColor, cx, cy; for (i = 0; i < a.length; i++){ eDist = dist(mouseX , mouseY , a[i][0], a[i][1]); eSize = map(eDist , 0, 200, 5, 100); eColor = map(eDist , 0, 200, 50, 255); fill(eColor , 200); cx = noise(mouseX)*10 + a[i][0]; cy = noise(mouseY)*10 + a[i][1]; ellipse(cx, cy , eSize , eSize); } } </pre>	<pre> float [][] a = new float[500][2]; void setup() { size(700, 500); for(int i = 0; i < a.length; i++){ for(int j = 0; j < a[i].length; j++){ a[i][j] = random(10,490); } } } void draw() { smooth(); noStroke(); background(0); for(int i = 0; i < a.length; i++){ float eDist = dist(mouseX, mouseY, a[i][0], a[i][1]); float eSize = map(eDist, 0, 200, 5, 100); float eColor = map(eDist, 0, 200, 50, 255); fill(eColor, 200); float cx = noise(mouseX)*10 + a[i][0]; float cy = noise(mouseY)*10 + a[i][1]; ellipse(cx, cy, eSize, eSize); } } </pre>

J'avais tiré le sketch en Java du livre « Programming for Artists » de Paul Orlov.

NB : Ce livre en russe est librement téléchargeable sur le [blog de Paul Orlov](#).

J'avais consacré une petite étude aux différences entre sketchs Processing et P5, dans laquelle j'avais utilisé le sketch de Paul Orlov, ainsi que quelques autres. On peut télécharger librement cette étude ici :

<https://github.com/gregja/p5Migration>

A.4 Les fonctions en JS

Tout au long de ce tutoriel, nous avons utilisé des fonctions Javascript. Certaines de ces fonctions étaient fournies par le langage JS lui-même, d'autres étaient fournies par P5. Ce n'est d'ailleurs pas toujours évident, quand on débute, de savoir quelle est la provenance d'une fonction. S'agit-il d'une fonction standard du langage, d'une fonction fournie par un framework complémentaire ? Beaucoup de développeurs débutants s'initient au JS en passant par le framework JS JQuery. Je trouve cela regrettable, car cela entraîne beaucoup de confusion dans l'esprit des développeurs débutants. C'est la raison pour laquelle j'ai privilégié dans le présent tutoriel l'étude du Javascript dit « natif », que l'on appelle aussi « Vanilla JS ». Et vous noterez que j'ai essayé de préciser, pour chaque fonction utilisée, s'il s'agissait d'une fonction JS ou d'une fonction P5.

Vous pouvez vous aussi créer des fonctions JS. Pour ce faire, il existe deux formes d'écriture :

- la première :

```
function mafonction(param1, param2) {  
    var toto = 3 ; // variable locale à la fonction xxx  
    return param1 + param2 + toto ;  
}
```

- la seconde :

```
var mafunction = function(param1, param2) {  
    var toto = 3 ; // variable locale à la fonction xxx  
    return param1 + param2 + toto ;  
} ;
```

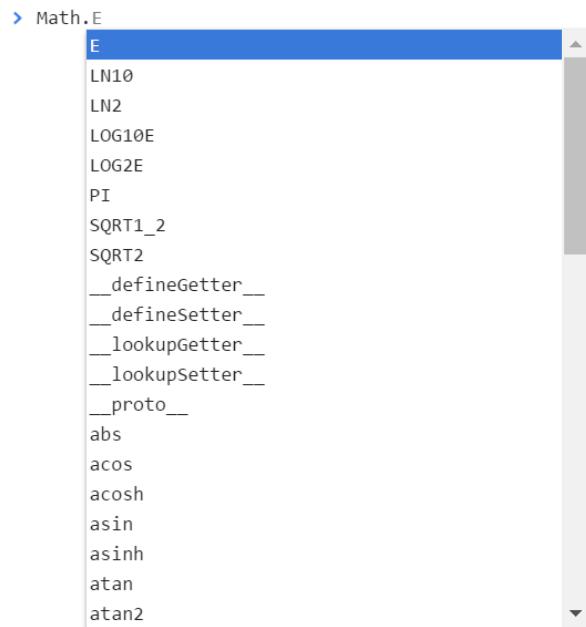
La première forme est une forme que tout le monde très bien, on la retrouve dans la plupart des langages de programmation.

Dans la seconde forme, on crée une variable et on lui affecte une fonction anonyme.

Il existe quelques différences subtiles entre ces 2 formes d'écriture, qu'il serait trop long de détailler ici. Au début de mon apprentissage du langage JS, j'étais quelque peu dérouté par la seconde forme, ayant l'habitude de pratiquer la première sur d'autres langages. Mais avec la pratique, j'ai finalement changé d'avis et j'avoue aujourd'hui une nette préférence pour la seconde forme.

Il faut savoir que JS recèle une multitude de fonctions, souvent cachées à l'intérieur d'objets. Par exemple, l'objet Math, qui est un objet standard de JS, contient un grand nombre de fonctions intéressantes. Pour en avoir un aperçu, allez dans la console de

votre navigateur, puis saisissez le mot clé « Math » suivi d'un point « . », vous devriez voir apparaître ceci :



On voit que l'objet Math contient une propriété PI, des fonctions comme « abs » qui renvoie la valeur absolue d'un nombre, des fonctions bien connues « sin » et « cos », et plein d'autres encore. Si l'on souhaite respecter la terminologie propre à la programmation objet, on parlera de méthodes plutôt que de fonctions. Donc dans le cas de l'objet Math, on dit que « abs », « acos » et les autres sont des méthodes appartenant à cet objet.

Tiens, c'est marrant, il n'existe pas de méthodes sur l'objet Math, pour la conversion de degrés en radians, et inversement. Or P5 fournit ces méthodes, c'est donc le signe qu'elles sont programmées dans P5. Je vous propose de créer ces 2 fonctions de conversion, histoire de nous entraîner à créer des fonctions, et en même temps de voir quelles formules sont utilisées ici.

Puisque P5 nous fournit une fonction « radians » (pour la conversion de degrés en radians), et une fonction « degrees » (pour la conversion inverse), j'ai honteusement « repompé » le code de P5, pour créer les deux fonctions que voici :

```
var radians = function(x) {
    return 2 * Math.PI * x / 360;
};

var degrees = function(x) {
    return 360 * x / (2 * Math.PI);
};
```

Vous pouvez vous amuser à créer ces fonctions dans la console du navigateur, et exécuter tout de suite après, une boucle d'alimentation d'un tableau « datas ». Voici ce que cela pourrait donner :

```
> var radians = function(x) {
    return 2 * Math.PI * x / 360;
};

var degrees = function(x) {
    return 360 * x / (2 * Math.PI);
};

< undefined

> var datas = [] ; // tableau de coordonnées vide
var ray = 100 ; // rayon du cercle
for (var i = 0; i <= 360; i+= .5) {
    var angle = radians(i);
    var x = ray * Math.cos(angle);
    var y = ray * Math.sin(angle);
    datas.push({x:x, y:y});
}
datas.length;
< 721
> |
```

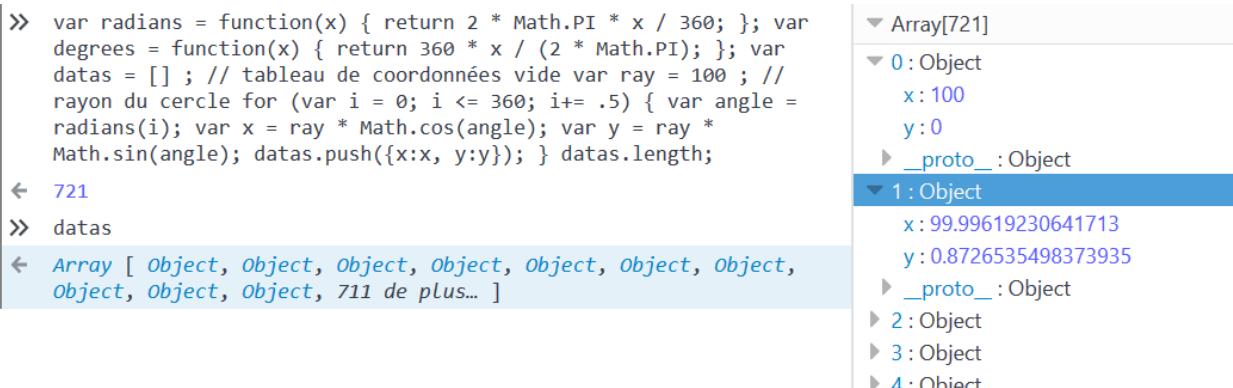
Vous noterez qu'à la fin du script, l'interpréteur JS affiche la valeur 721, cela correspond au nombre de « postes » (ou « éléments ») qui ont été créés dans ce tableau par notre boucle « for ».

Si vous tapez maintenant le nom du tableau (datas) dans la console du navigateur, vous verrez que vous pouvez vous amuser à « déplier » le tableau en cliquant sur les petites flèches.

```
> datas
< ▼Array[721] ⓘ
  ▼ [0 ... 99]
    ▼ 0: Object
      x: 100
      y: 0
      ► __proto__: Object
    ▼ 1: Object
      x: 99.99619230641713
      y: 0.8726535498373935
      ► __proto__: Object
    ▼ 2: Object
    ▼ 3: Object
    ▼ 4: Object
    ▼ 5: Object
```

Chrome affiche les postes du tableau par paquets de 100 éléments (0 à 99, 100 à 199, etc...). C'est un peu déroutant au début, mais on s'y habitue.

Dans la console de Firefox, on retrouve le même niveau d'information, mais présenté différemment :



The screenshot shows the Firefox Developer Tools Console with the following code and output:

```
> var radians = function(x) { return 2 * Math.PI * x / 360; }; var degrees = function(x) { return 360 * x / (2 * Math.PI); }; var datas = [] ; // tableau de coordonnées vide var ray = 100 ; // rayon du cercle for (var i = 0; i <= 360; i+= .5) { var angle = radians(i); var x = ray * Math.cos(angle); var y = ray * Math.sin(angle); datas.push({x:x, y:y}); } datas.length;
< 721
> datas
← Array [ object, object, object, object, object, object, object, object, object, 711 de plus... ]
```

A sidebar on the right shows the array structure:

- Array[721]
- 0 : Object
 - x : 100
 - y : 0
 - __proto__ : Object
- 1 : Object (highlighted)
 - x : 99.99619230641713
 - y : 0.8726535498373935
 - __proto__ : Object
- 2 : Object
- 3 : Object
- 4 : Object

Il faut noter que le langage JS évolue. Ce langage s'appuie sur la norme ECMAScript, et en 2015 a été publiée une version 6 de cette norme (généralement désignée sous le nom de « ES6 »... mais aussi quelquefois sous le nom de « ES2015 »). Cette nouvelle version embarque un grand nombre de nouveautés. Parmi ces nouveautés, on trouve les fonctions fléchées (en anglais « arrow functions »). Voici comme écrire nos 2 fonctions « radians » et « degrees » avec cette nouvelle syntaxe :

```
var radians = (x) => 2 * Math.PI * x / 360;
var degrees = (x) => 360 * x / (2 * Math.PI);
```

Cette manière de déclarer les fonctions est de plus en plus utilisée, vous la rencontrerez forcément quand vous lirez le code d'autres développeurs.

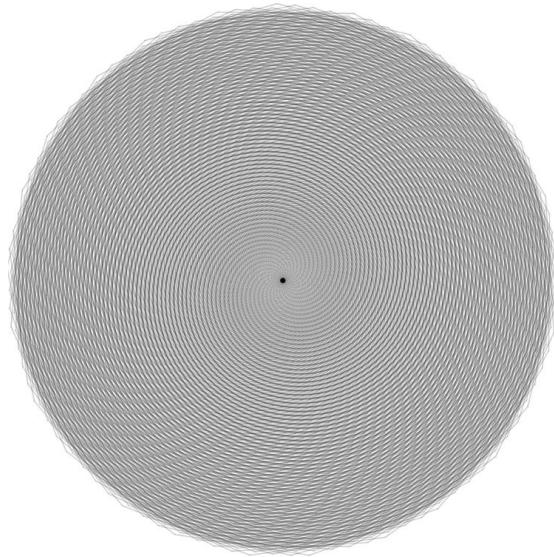
Il existe une différence subtile entre la déclaration des fonctions fléchées et les fonctions créées de manière plus... traditionnelle. Je pense que cela compliquerait inutilement les choses, que de vous présenter cette différence maintenant (ce n'est pas vital à notre niveau).

Si néanmoins le sujet vous intéresse, je vous recommande la page suivante, où le sujet est très détaillé :

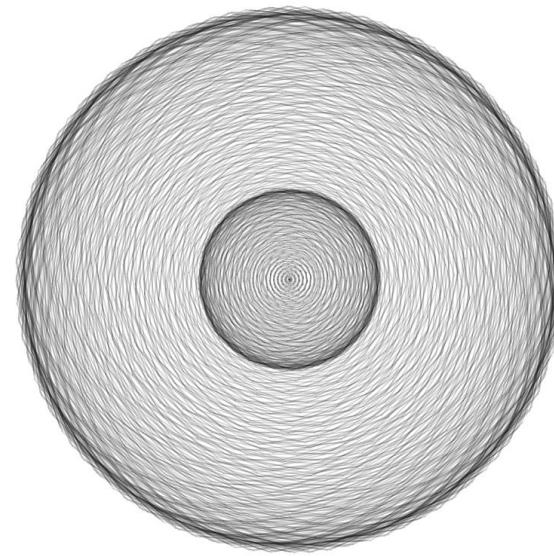
https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Fonctions/Fonctions_f1%C3%A9ch%C3%A9es

A.5 Expérimentations

A partir d'une version de travail du sketch étudié au chapitre 3.1.1, voici l'image obtenue pour la formule suivante : $x * x * \sin(1 / x)$



Et une autre image obtenue avec la formule suivante : $1 + 2 * \cos(2 * x)$



```

"use strict";
var b = 500; var sx = 500; var sy = 500;
var datas, datas_length, datas_i;

var fna = function(x) {
    return x * x * sin(1 / x);
//  return 1 + 2 * cos(2 * x);
}

function setup() {
    createCanvas(sx, sy);
    stroke(0, 50);

    var hx = sx / 2;
    var hy = sy / 2;

    var z, r;
    var m = 1.0e-30;
    console.log(m);
    for (z=0 ; z <= TWO_PI ; z+=.1) {
        r = abs(fna(z));
        if (m < r) {
            m = r + 0.1;
        }
    }
    console.log(r);
    console.log(m);

    var u, v;
    datas = [];
    for (z=0 ; z <= TWO_PI ; z+=.001) {
        r = abs(fna(z));
        u = hx + hy * cos(b * z) * r / m;
        v = hy + hy * sin(b * z) * r / m;
        if (v < 0 || v > sy) {
            console.log('en dehors des limites');
        } else {
            datas.push({x:u, y:v});
        }
    }
    datas_length = datas.length;
    if (datas.length == 0) {
        datas_i = 0;
        noLoop();
    } else {
        datas_i = 1;
    }
}

function draw() {
    line(datas[datas_i-1].x, datas[datas_i-1].y,
          datas[datas_i].x, datas[datas_i].y);
    datas_i += 1;
    if (datas_i >= datas_length) {
        noLoop();
    }
}

```

```
function keyPressed() {  
  if (key == 'x' || key == 'X') {  
    noLoop();  
  }  
}
```