
# Génération procédurale

Filière informatique 1ère année
Projet informatique S6

**Groupe 12828**

***Auteurs :***
Arnaud GRAPPE
Aymane LAMHAMDI
Yves LEBARBIER
Dylan MACHADO

***Encadrant :***
M. Christophe SCHLICK

***Directeur de projet :***
M. David RENAULT

# Table des matières

**I Présentation du sujet** **3**
I.1 Objectif du projet . . . . . 3
I.2 Description détaillée du sujet . . . . . 3

**II Cadre de travail et organisation de l'équipe** **5**
II.1 Cadre de travail . . . . . 5
II.2 Organisation . . . . . 5
II.3 Organisation du dépôt . . . . . 5

**III Architecture logicielle** **6**
III.1 Structure des textures . . . . . 6

**IV Générateurs** **8**
IV.1 Structure et fonctionnement global . . . . . 8
IV.2 Pavages réguliers . . . . . 8
IV.3 Générateurs de bruit de gradient . . . . . 9
IV.4 Interpolation à base de distances signées . . . . . 11
IV.5 Diagramme de Voronoi . . . . . 13

**V Filtres** **14**
V.1 Structure et fonctionnement global . . . . . 14
V.2 Filtres de composition . . . . . 14
V.3 Filtres de transformations colorimétriques . . . . . 15

**VI Résultats et retour d'expérience** **15**

# Table des figures


1 Exemples d'images générées au cours de ce projet . . . . . 4

2 Structure du dépôt . . . . . 6

3 Génération d'un pavage carré . . . . . 8

4 Décomposition d'un pavage hexagonal en un pavage rectangulaire . . . . . 9

5 Bruit de Perlin . . . . . 10

6 Bruit fractal . . . . . 11

7 Interpolation avec une distance signée basique . . . . . 12

8 Images obtenues avec les fonctions de distance signées . . . . . 12

9 Deux variantes du diagramme de Voronoi . . . . . 14

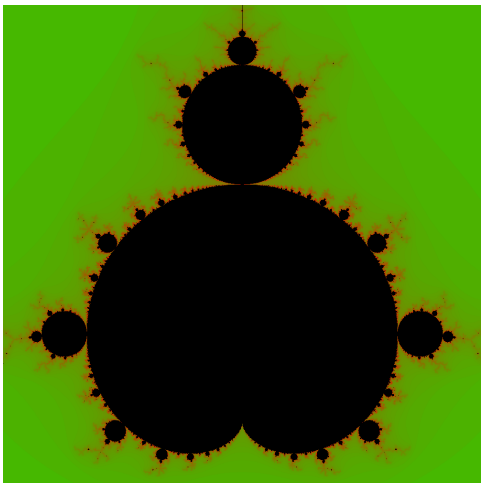
# I Présentation du sujet

## I.1 Objectif du projet

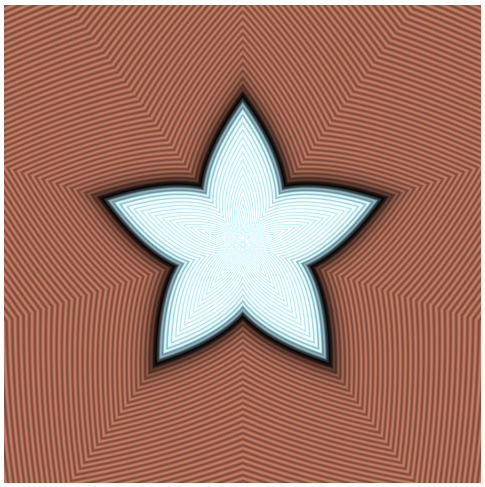
Grâce à l'apparition de l'informatique, l'être humain s'est souvent intéressé (et continue de l'être) à automatiser certaines tâches puisque une machine est plus compétente dans le calcul que l'Homme. Il existe divers domaines, tels que l'industrie, dans lesquels l'automatisation a beaucoup d'utilité et dans ce projet, nous nous intéressons à la génération procédurale qui consiste à créer du contenu numérique (qui, dans ce cadre, sera des images en 2D) en grande quantité et de manière automatisée devant répondre un certain nombre de règles définies par des algorithmes. Autrement dit, la génération procédurale est une méthode qui consiste à produire des objets grâce aux algorithmes afin d'éviter de les faire manuellement. Ce projet aura pour objectif de mettre en place des bibliothèques comportant des fonctions qui permettront de générer, transformer et composer des textures. Chaque fonction doit être indépendante des autres et le but final sera de créer une interface dynamique pour générer une texture en fonction des paramètres proposés à l'utilisateur.

L'image 1 montre quelques exemples de résultats obtenus grâce à la génération procédurale.

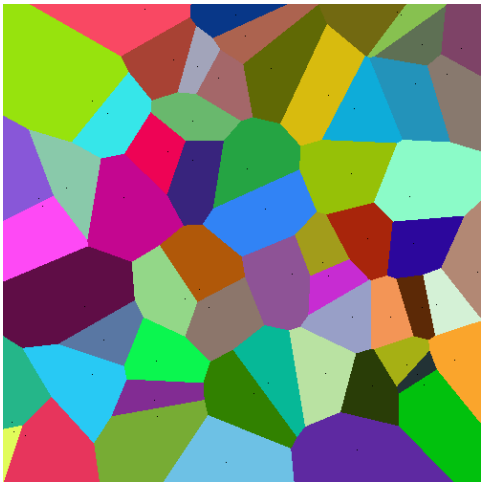
## I.2 Description détaillée du sujet



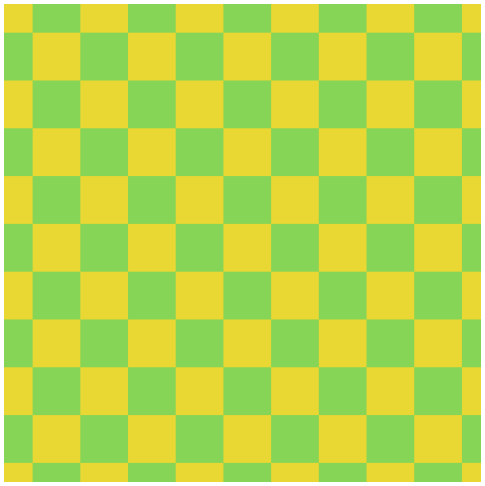
(a) Ensemble de Mandelbrot



(b) Avec une distance signée



(c) Diagramme de Voronoi



(d) Pavage régulier

FIGURE 1 – Exemples d'images générées au cours de ce projet

# II Cadre de travail et organisation de l'équipe

## II.1 Cadre de travail

La réalisation de ce projet a débuté le mardi 9 mars et elle s'est achevée le vendredi 21 mai. Contrairement au projet *Tilings* du premier semestre, nous avons pu nous réunir chaque semaine dans une salle informatique de l'ENSEIRB. Cette organisation nous a permis d'interagir plus facilement entre nous comme avec nos encadrants.

## II.2 Organisation

Au début de chaque séance, nous passions en revue les réalisations de la semaine dernière. Cela nous permettait de définir les objectifs pour la séance, et de nous accorder sur une répartition du travail à effectuer. Nous avions commencé d'abord par définir notre noyau et le fixer et par organiser nos fichiers. En parallèle, nous avions commencé par implémenter certains générateurs et filtres.

## II.3 Organisation du dépôt

Pour les besoins du projet, la figure 2 montre l'organisation des fichiers dans le dépôt.
Le Makefile contient une règle test permettant d'exécuter les tests propres au projet. Le répertoire `src/` contient les fichiers sources du projets. Le répertoire `doc/` contient la documentation du projet générée par JSDoc [1].

1. JSDoc est un langage de balisage utilisé pour documenter les codes sources Javascript. https ://jsdoc.app/

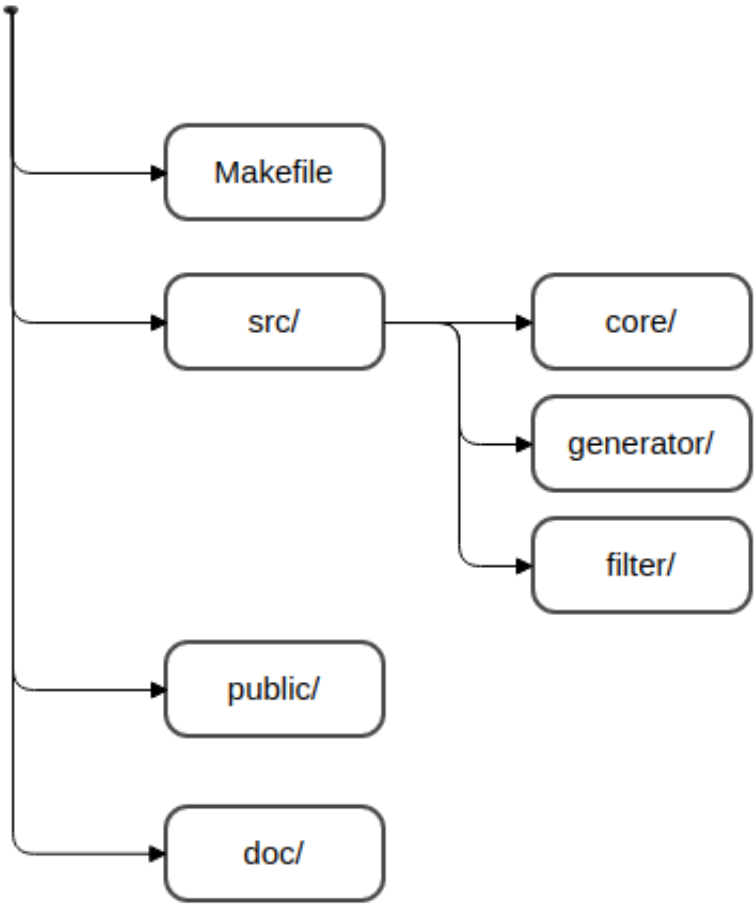



FIGURE 2 – Structure du dépôt

# III Architecture logicielle

Afin de pouvoir créer différents générateurs et filtres et de pouvoir les composer entre eux, il est nécessaire que la structure de ceux-ci et des fonctions gérant l'affichage soit clairement définie et soit commune à tout le projet.

## III.1 Structure des textures

Une texture est un ensemble de générateurs et de filtres organisés afin de produire une image. Nous avons décidé de visualiser les textures sous forme d'arbre, permettant une implémentation simple et efficace. Dans ces arbres, les nœuds internes représentent des filtres et les feuilles sont des générateurs. Nous avons donc implémenté pour ceci 2 structures : d'abord les listes puis

les arbres. En effet les enfants d'un noeud dans l'arbre sont contenu dans une liste, nous avons donc besoin de cette structure.

### Les listes

Nous avons donc implémenté des listes respectant les bases de la programmation fonctionnelle c'est-à-dire qu'elle sont en fait des paires pointées composées d'un tête et d'une queue, où la tête est la première valeur de la liste et la queue est le reste de la liste donc elle même une liste. Il existe de plus la liste vide `nil={}`.
Afin de gérer ces listes, nous avons implémenté différentes fonctions :

- la fonction `cons` qui étant donnés deux paramètres, crée la liste ayant pour tête le premier paramètre et pour queue le deuxième,
- la fonction `head` qui permet d'accéder à la tête d'une liste passée en paramètre,
- la fonction `tail` qui permet d'accéder à la queue d'une liste passée en paramètre ;
- la fonction `isEmpty` qui teste si la liste donnée en paramètre est vide et renvoie `True` le cas échéant.

### Les arbres

Pour les arbres, nous avons fonctionné de la même manière. En effet, les arbres sont des paires pointées, composées de la valeur de la racine et de la liste des enfants, ces enfants étant eux-mêmes des arbres. La liste vide `nil` précédemment définie est aussi considérée comme l'arbre vide.
Comme pour les listes, différentes fonctions gèrent la création et l'utilisation des arbres :

- la fonction `node`, équivalent de `cons` pour les arbres, qui crée donc un arbre avec pour racine le premier paramètre et pour enfants le deuxième,
- la fonction `val`, qui permet d'accéder à la valeur de la racine de l'arbre passé en paramètre ;
- la fonction `children`, qui renvoie la liste des enfants de l'arbre passé en paramètre.

# IV Générateurs

Les générateurs sont la base de la création d'image, en effet ce sont eux qui créent une image à partir de rien.

## IV.1 Structure et fonctionnement global

Nous avons implémenté les générateurs comme des fonctions prenant en paramètres les différents options gérant le générateur, comme la taille de l'image ou le nombre de motifs. Ces options sont données dans un dictionnaire nommé `args` qui comporte donc différentes clés, `width` par exemple.
Les générateurs fonctionnent tous de la même manière, ils créent une image au format `imageData` du module `canvas` puis ils la remplissent pixel par pixel en fonction du générateur. Une fois l'image remplie, elle est renvoyée puis traitée par la fonction gérant les textures.

## IV.2 Pavages réguliers

Les pavages réguliers présentent une classe de textures qui se prête assez bien à la génération procédurale. Notamment, les pavages plus simples (carré, triangulaire) peuvent être dessinés simplement par un unique parcours de tableau. Dans le cas d'un pavage carré bicolore par exemple, pour un couple de coordonnées $(x, y)$, il suffit de calculer les modulo de $x$ et $y$ par le côté du carré. La somme de ces deux modulos nous donne un entier dont la parité

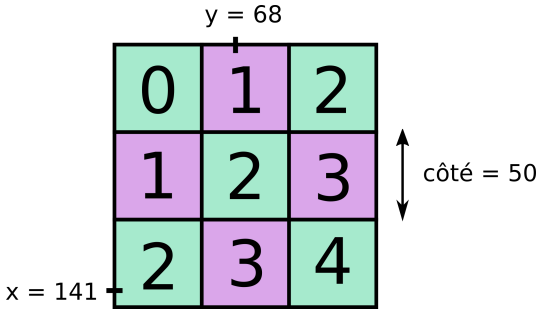



FIGURE 3 – Génération d'un pavage carré

détermine si le couple $(x, y)$ appartient à un carré bleu ou à un carré violet. Dans la figure 3, on a $x\%50 + y\%50 = 3$, le couple $(x, y)$ correspond donc à un carré violet.

Le processus est déjà plus compliqué pour la création d'un pavage hexagonal. On peut par exemple découper un pavage hexagonal en rectangles, ce

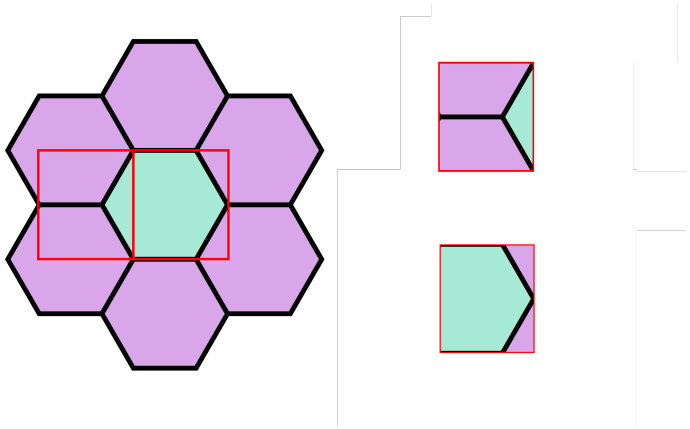


FIGURE 4 – Décomposition d'un pavage hexagonal en un pavage rectangulaire

qui nous permet de réutiliser le code produit pour la génération de pavages carrés (voir figure 4).

Se pose ensuite le problème de la couleur : pour chaque rectangle, il faut pouvoir déterminer si on se situe dans une zone bleue ou violette... Toute relative qu'elle soit, la complexité de cette opération nous a poussé à tenter une autre approche : l'approche descendante choisie pour notre structure de textures nous permet de générer un pavage polygone par polygone.
Le fichier `src/generators/polygons/polygons.impl.js` implémente les objets *point* et *polygon*, ainsi qu'une fonction `pointIsIn` qui détermine si un point se situe à l'intérieur d'un polygone. Cette fonction utilise un algorithme dit de *ray casting*, et dont le code provient d'un forum sur Stack Overflow [2]

## IV.3 Générateurs de bruit de gradient

Les générateurs basés sur le bruit de gradient créent une image en interpolant des valeurs pseudo-aléatoires attribuées sur des cases prédéfinies de celle-ci.

### Bruit de Perlin

Le bruit de Perlin est un bruit de gradient particulier. Dans le générateur de ce bruit, pour chaque pixel, on définit la case dans laquelle il se trouve, qui est définie par une découpe de l'image en $p \times p$ rectangles, $p$ étant un paramètre du générateur. Ensuite, 4 produits scalaires sont calculés, un pour chaque coin de cette case. Pour un coin en particulier, c'est le produit scalaire

2. https ://stackoverflow.com/questions/217578/how-can-i-determine-whether-a-2d-point-is-within-a-polygon

du vecteur distance du pixel à ce coin avec un vecteur pseudo-alétoire associé au coin. Ce vecteur pseudo-aléatoire vaut $(\pm 1, \pm 1)$, où les signes sont décidés par la valeur du pixel et une permutation de $[1, 256]$ appliquée 2 fois, ce qui permet d'avoir un résultat qui parait aléatoire. Une fois les 4 produits scalaires calculés, ils sont interpolés afin d'obtenir une valeur entre -1 et 1, qui est ensuite ramenée entre 0 et 1, ce qui donne la valeur finale du pixel. Pour obtenir une image avec cette valeur, on effectue pour chaque pixel une dernière interpolation linéaire entre deux couleurs données en paramètre avec la valeur du pixel, comme le noir et le blanc dans l'exemple Figure 5.

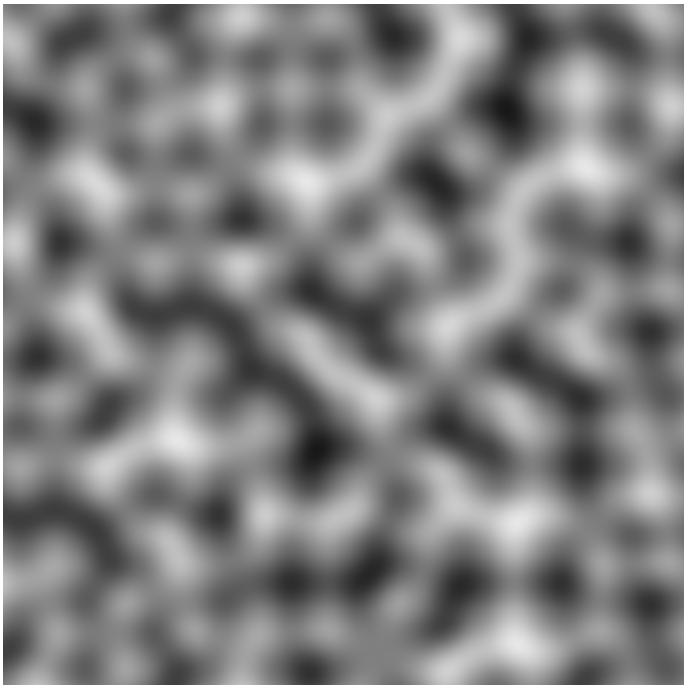


FIGURE 5 – Bruit de Perlin

### Bruit fractal

Une forme plus avancée de bruit de gradient est le bruit fractal. Celui-ci utilise le bruit de Perlin précédemment défini afin de créer des textures plus réalistes. Ce générateur consiste à ajouter plusieurs bruits en augmentant à chaque fois la fréquence et en diminuant l'amplitude. En pratique, pour un nombre d'itérations donné par le paramètre `p` du générateur, on calcule un bruit de Perlin de fréquence $2^{k+1}$ et d'amplitude $2^{-(k+1)}$ c'est-à-dire que la découpe de l'image du bruit de Perlin donne 4 fois plus de cases et que le résultat et multiplié par une valeur 2 fois plus petite à chaque itération. Avec

ce procédé, on obtient une image qui s'apparente à des textures plus réalistes que le bruit de Perlin simple, on peut en voir l'exemple Figure 6

FIGURE 6 – Bruit fractal

## IV.4 Interpolation à base de distances signées

Les fonctions de distance signées[3] sont des fonctions qui associent à chaque point de l'espace une valeur de distance qui peut donc être positive ou négative. On peut alors choisir des fonctions de ce type qui forment des formes intéressantes et on peut visualiser celles-ci en donnant une couleur particulière aux points donc la valeur de distance est nulle. Un exemple simple est de simplement utiliser une distance euclidienne au centre de l'image et de lui retirer une valeur arbitraire (dans notre cas la valeur est donnée par le paramètre `r` de la fonction). Cet exemple est illustré Figure 7.

Sur cette figure, on observe bien le cercle noir, qui correspond aux pixels où la distance vaut 0. De plus, une fonction périodique a été appliquée à la distance afin d'obtenir les cercles concentriques que l'on voit sur l'image.

3. En anglais Signed Distance Functions

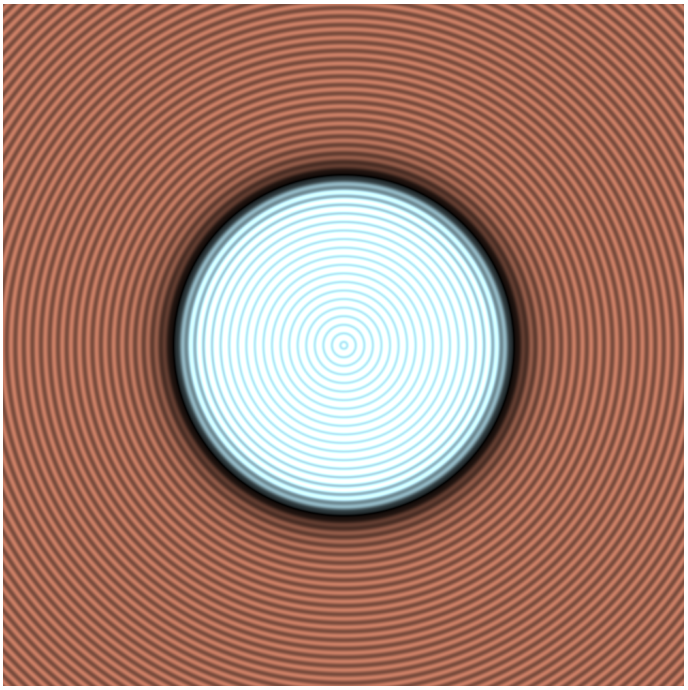


FIGURE 7 – Interpolation avec une distance signée basique

Ensuite, pour générer des images plus complexes, il faut construire les fonctions de distances signées correspondantes. Deux exemples sont visibles Figure 8.

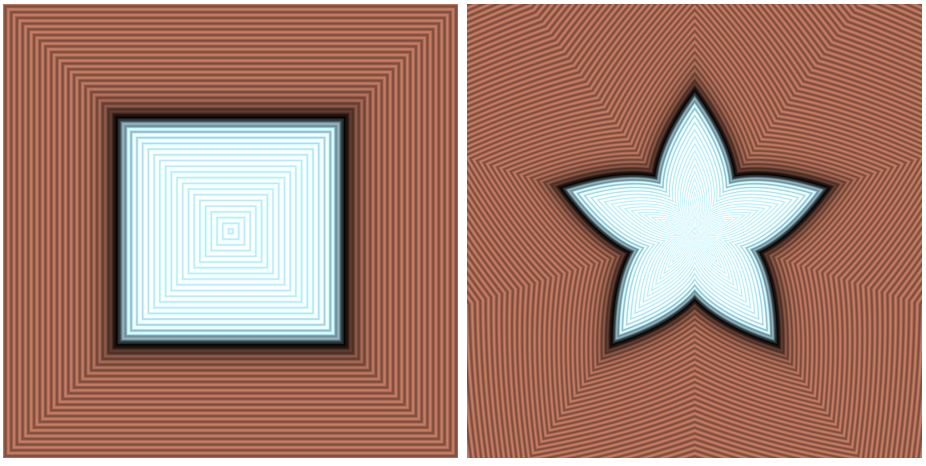


FIGURE 8 – Images obtenues avec les fonctions de distance signées

Pour le rectangle, on utilise une distance analogue à la norme 1 : $|(x, y)|_1 =$

$max(|x|, |y|)$. En appliquant des coefficients aux deux coordonnées et en retirant une valeur arbitraire comme pour le cercle, on obtient ce résultat.

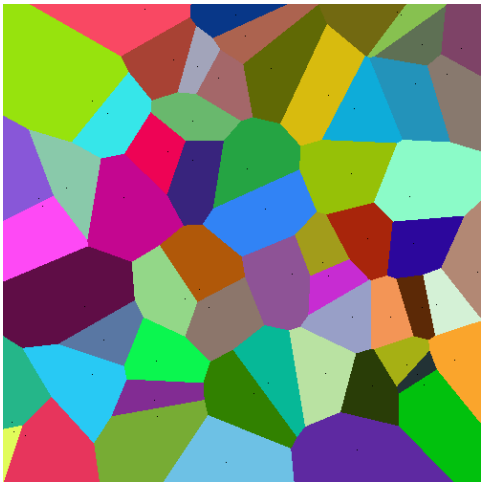
Pour l'étoile, on découpe le plan en $n$ sections, `n` étant un paramètre du générateur, afin d'avoir une branche de l'étoile par section. On calcule donc pour chaque pixel son angle à l'aide d'une arc-tangente afin de déterminer dans quelle section il se trouve. Une fois cette information connue la valeur qui lui est associée est sa distance à l'origine moins une valeur dépendant de son angle à l'intérieur de la section, ce qui va créer les bords de l'étoile et aboutir à l'image finale.

## IV.5 Diagramme de Voronoi

Le diagramme de Voronoi [4] est aussi appelé partition ou décomposition de Voronoi. En effet, de manière plus générale, il représente une décomposition d'un espace métrique en cellules (autrement dit en régions adjacentes) déterminée par les distances à un ensemble discret d'objets de l'espace, généralement un ensemble discret de points [5]. Dans notre cas, nous travaillons dans un plan et les cellules seront donc des polygones appelés polygones de Voronoi. De façon plus détaillée, le diagramme de Voronoi est un pavage du plan en cellules dont chacune d'entre elles est obtenue en fonction de la distance à un point particulier appelé germe. Chaque cellule représente en quelque sorte la *zone d'influence* du germe.
Le principe de l'algorithme pour obtenir ce diagramme est très simple : on commence d'abord par initialiser les germes en attribuant à chacun une couleur puis on parcourt la matrice de pixels et on calcule la distance euclidienne du pixel courant à tous les germes et enfin on le colorie en la couleur du germe correspondant à la distance minimale. La figure 9a présente le diagramme de Voronoi obtenu en appliquant cet algorithme.

Pour aller plus loin, on peut réaliser à partir du diagramme de Voronoi une animation d'une texture dite de *peau de reptile* comme le présente la figure 9b. Mis à part la texture des germes, il existe deux différences importantes entre ces deux variantes. D'une part, il y a l'ajout des bordures scindant ainsi les différentes zones d'influence adjacentes et d'autre part le contour de chaque zone est reproduit de manière concentrique.

4. Mathématicien russe de la fin du XXème siècle
5. Source : *Wikipédia*



(a) Diagramme de Voronoi

(b) Avec une texture de reptile

FIGURE 9 – Deux variantes du diagramme de Voronoi

# V Filtres

Une fois des générateurs implémentés, des filtres peuvent leur être appliqués, afin de modifier l'image, de l'analyser, ou de faire la composition de plusieurs images.
Les différents filtres ont été implémentés sur le fichier `filters.impl.js` du répertoire `src/filters/`.

## V.1 Structure et fonctionnement global

Dans notre implémentation, les filtres sont des fonctions qui, comme les générateurs, prennent leurs différentes options en paramètres dans un dictionnaire `args`. Cependant ils prennent aussi en paramètre un tableau d'images au format renvoyé par les générateurs, sur lesquelles ils vont agir.
Au niveau du fonctionnement, les filtres réécrive sur la première image donnée en paramètre et le font pixel par pixel. Après que tous les pixels aient été traités, l'image est renvoyée et gérée ensuite par la fonction appliquant les textures.

## V.2 Filtres de composition

Les différents filtres de composition de deux images implémentées sont :

— filtre de division : **divide**

— filtre d'addition : **add**

— filtre de soustraction : **substract**

— filtre de multiplication : **multiply**

### V.3 Filtres de transformations colorimétriques

Les différents filtres de transformations colorimétriques implémentés sont :

— filtre d'opacité : **dark**

— filtre de luminosité : **luminosity_up** et **luminosity_down**

— filtre de contraste : **contrast_up** et **contrast_down**

— filtre d'inversion : **reverse**

— filtre de niveaux de gris : **grayscale**

— filtre de floutage : **blurry** et **blurryGenerate**

— filtre de translation : **translateRight** et **translateLeft**

— filtre de multiplication : **translateTorusRight** et **translateTorusLeft**

## VI Résultats et retour d'expérience

Dans le cadre des projets de programmation, nous avons donc réalisé une interface permettant de générer des textures en fonction des paramètres sélectionnés. Ce projet nous a permis de mettre en pratique les notions vues en programmation fonctionnelle telles que la structure de liste et d'arbre ainsi que la notion de première classe pour les fonctions. Il nous a de plus offert l'occasion d'apprendre à travailler en effectifs plus conséquents en comparaison avec le projet *Tilings* mais aussi à découvrir quelques éléments du language `HTML` tels que le canvas.
Par ailleurs, tout au long de ce projet, nous avons rencontré de nombreuses notions mathématiques concernant la génération de texture ce qui aide à développer notre capacité à comprendre et appliquer les théories en code informatique.
Aussi, notre professeur encadrant nous a donné de précieux conseils et nous l'en remercions.