Institut Supérieur
d'Informatique
de Modélisation
et de leurs Applications

complexe des cézeaux
BP 125 - 63173 aubière cedex

Rapport de Projet 3ème année

Synthèse de textures en C avec interface Matlab

Présenté par : Lahcen Mouzoun
et Gwénolé Quellec
Responsable : Pierre Chainais

Projet de 200 heures
de Novembre 2004 à Mars 2005

Résumé

Les **Cascades Infiniment Divisibles** sont un modèle stochastique invariant d'échelle récemment découvert. Ce modèle est particulièrement flexible. Nous avons étudié ce procédé dans le cas de la **synthèse de textures**. Des recherches sont toujours en cours sur le sujet, d'où la nécessité de disposer d'un outil performant permettant de valider différentes propriétés.

Notre travail a consisté à développer une boîte à outil complète, implémentée sous Matlab en langage C, munie d'une **interface graphique**. Afin d'augmenter la **rapidité d'exécution**, on a tout d'abord réimplémenté une fonction de génération de textures. Nous lui avons de plus ajouté de nouvelles fonctionnalités telles que la création de nouveaux motifs de texture. Puis pour valider la propriété d'invariance d'échelle, on a écrit des fonctions permettant d'effectuer des **zooms** dans une texture. Pour illustrer la souplesse de la méthode, on a également ajouté différentes fonctions pour **mélanger plusieurs textures**.

Mots clefs : Cascades Infiniment Divisibles, synthèse de textures, interface graphique, rapidité d'exécution, zooms, mélanger plusieurs textures

Abstract

Compound Poisson Cascade is a scale invariant stochastic process recently discovered. This process is particularly flexible. We studied the case of **texture generation**. Research on the subject is still under way hence the need of an efficient tool to validate different properties.

The work we carried out for this project consisted in developing a complete toolbox, implemented in C language under Matlab, fitted with a **graphical user interface**. To improve **computation times**, we first reimplemented a texture generation function. We added new functionalities such as new kinds of texture patterns. Then to validate the scale invariance property, we coded functions to **zoom** in a texture. To show the flexibility of the method, we also added different functions to **mix several textures**.

Keywords : Compound Poisson Cascade, texture generation, graphical user interface, computation times, zoom, mix several textures

Table des matières

Table des figures

Introduction

Dans le cadre de notre troisième année d'études en spécialité calcul scientifique et modélisation à l'ISIMA, nous avons réalisé un projet de deux cent heures réparties sur cinq mois, entre novembre 2004 et mars 2005.

Les Cascades Infiniment Divisibles, un modèle stochastique invariant d'échelle récemment découvert, est utilisé pour la synthèse de textures. Le but étant notamment de reproduire des images naturelles. C'est dans ce cadre que nous avons développé une boîte à outils permettant de faciliter la validation des différentes propriétés.

Dans un premier temps, nous allons présenter le principe de la synthèse ainsi que les différents algorithmes que nous avons implémenté. Puis nous détaillerons la composition du programme et les fonctionnalités de l'interface. Enfin, nous expliquerons comment faire évoluer le programme.

1 Description des algorithmes

1.1 Principe de la génération

On cherche à construire une matrice de texture, appelée Q_r, dans le plan $r = 0$. Pour cela on génère des points, appelés multiplieurs (m_i), au dessus du plan de la texture : entre les plans $r = 1$ et $r = r_{min}$ $(0 < r_{min} \leq 1)$ selon une distribution en $\frac{1}{r^2}$ $(p(m_i \in [r; r+dr]) = \alpha \times \frac{1}{r^2})$.

A chaque multiplicateur est associé une valeur ω_i tirée selon une distribution donnée. La matrice Q_r est construite de la manière suivante :

- on multiplie chaque élément e de la matrice par les ω_i de tous les multiplieurs m_i situés dans un cône au dessus de e
- ou de manière équivalente, on multiplie par ω_i chaque élément e de la matrice appartenant à la base d'un cône en dessous d'un multiplieur m_i (voir figure 1)

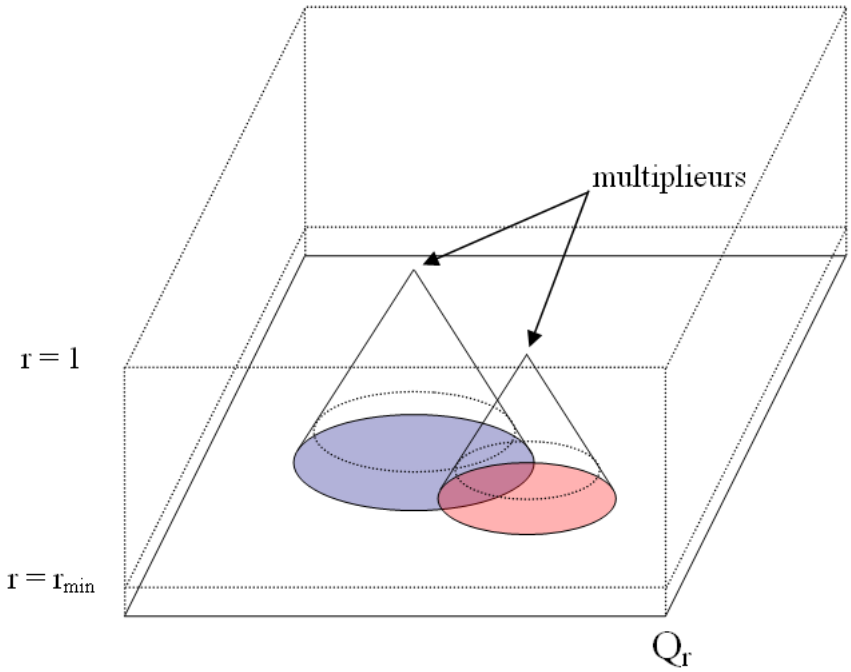

FIG. 1 – Principe de génération de la texture

1.2 Tirage des multiplieurs

Les lois de génération des multiplieurs implémentées sont les suivantes :

- la loi log-normal, de paramètres μ et σ^2
- la loi exponentielle, de paramètre la constante de temp τ
- la loi log-Poisson ou she-Leveque ou Dirac, de paramètre ω_0

Pour générer la texture, on tire des multiplieurs avec une densité C dans un pavé :

- délimité par les altitudes r_{min} et r_{min_prec}, la nouvelle et la précédente résolution de la cascade
- de base $[x - \frac{r_{min\_prec}}{2}; x + \Delta x + \frac{r_{min\_prec}}{2}] \times [y - \frac{r_{min\_prec}}{2}; y + \Delta y + \frac{r_{min\_prec}}{2}]$, les points de la matrice Q_r à calculer étant compris dans le carré $[x; x + \Delta x] \times [y; y + \Delta y]$.

NB : lors de la première génération, r_{min_prec}=1, $x = y = 0$, $\Delta x = x_{max}$ et $\Delta y = y_{max}$.

Selon les directions x et y, les coordonnées x_i et y_i sont tirées uniformément. Selon l'axe vertical, on tire uniformément une valeur u_i, puis on calcule $r_i = \frac{1}{u_i^2}$.

En fonction de la position (x_i, y_i, r_i) du multiplicateur m_i, on détermine les points de la matrice Q_r susceptibles d'être modifiés par m_i. Cette zone est un carré de côté r_i centré en

(x_i, y_i) restreint à $[x; x + \Delta x] \times [y; y + \Delta y]$, appelé zone d'influence de m_i. Si cette zone est vide, on élimine le multiplieur.
Pour les multiplieurs qui restent :

- on tire un multiplicateur ω_i selon la loi de génération :
  - $\omega_i = exp(\sqrt{\sigma^2} * randn() + \mu)$ pour la loi log-normal
  - $\omega_i = ((1 + \tau)^{\frac{1}{\tau}} * randu())^{\tau}$ pour la loi exponentielle
  - $\omega_i = \omega_0$ pour la loi log-Poisson
- on leur affecte les informations sur le motif et la zone de la texture (voir paragraphe 1.5) à laquelle ils appartiennent
- on tire un angle de rotation du motif uniformément sur $\theta_i = [0; 2\pi]$.

$randu()$: tirage d'un nombre suivant une loi uniforme sur [0 ;1[
$randn()$: tirage d'un nombre suivant une loi normale centrée réduite.

1.3 Générateurs de nombres aléatoires utilisés

Comme on l'a vu au paragraphe précédent, on a besoin d'un générateur de nombres suivant une loi uniforme et d'un générateur de nombres suivant une loi normale.

Le générateur de nombres uniforme de la librairie standard du C est à valeurs dans $\{0, 1, ..., 2^{16}\text{-}1 = 65.535\}$, ce qui est trop faible. On a donc utilisé un autre générateur : *Mersenne twister* (cf. références [1] et [2]). Ce générateur a une période de $2^{19937} - 1$ et est, contrairement à d'autres générateurs, indépendant du système.

Pour générer des nombres suivant une loi normale, on utilise le générateur uniforme précédent et on utilise la forme polaire de la transformation de Box-Muller (cf. référence [3]) : on génére deux nombres suivant une loi normale à partir de deux nombres suivant une loi uniforme.

1.4 Construction de la matrice Q_r

1.4.1 Méthode

Pour générer la matrice Q_r, on s'est demandé s'il valait mieux parcourir les éléments de Q_r et chercher les multiplieurs qui l'*éclairent* (1) ou parcourir les multiplieurs et considérer les éléments de Q_r qu'ils *éclairent* (2).

Les éléments de la matrice Q_r sont ordonnés dans l'espace (dans le plan) selon x et y. Il n'est pas possible d'en faire de même pour l'ensemble des multiplieurs $M = \{(x_i, y_i, r_i, ...)/i \in \{1, .., m\}\}$. Ainsi, pour trouver les points contenus dans un cône au dessus d'un élément de Q_r, il faut parcourir tous les éléments de M. Alors que pour chaque multiplieur, on a déterminé une zone d'influence : on n'a pas besoin de parcourir les éléments de Q_r qui en sont à l'extérieur.
Soit :

- $n = |Q_r|$ le nombre de points de discrétisation dans la matrice Q_r
- $m = |M|$
- $z_i \leq m$ le nombre de points de discrétisation dans la zone d'influence de $m_i \in M$

Dans le cas (1), le nombre de points à parcourir est $n \times m$. Dans le cas (2), le nombre de points à parcourir est borné par $max(z_i) \times n$. Dans tous les cas, l'alternative (2) est plus avantageuse.

Pour chaque type de motif, on a écrit une fonction qui recherche les éléments de Q_r éclairés par un multiplieur m_i donné. Ces éléments sont alors mis à jour.

Lorsque l'on construit une matrice Q_r, on parcourt donc la liste des multiplieurs et on appelle la fonction qui correspond au motif. Il faut la distinction entre deux types de motifs : les motifs qui définissent la base d'une pyramide et les fonctions lissantes :

- Dans le premier cas, on part avec une matrice $Q_r = 1$. La mise à jour d'un élément se fait en multipliant sa valeur par ω_i.
- Dans le cas des fonctions lissantes, on part avec une matrice $Q_r = 0$. La mise à jour d'un élément e_j se fait en ajoutant $log(\omega_i) \times f(m_i, e_j)$ à sa valeur. Puis on prend l'exponentielle de Q_r.

Lorsque l'on construit une matrice Q_r à partir de points $m_i^l = (x_i^l, y_i^l, r_i^l, w_i^l, id\_fonction_i^l, ...)$ portant un motif utilisant une fonction lissante mélangés avec d'autres points $m_i^{nl} = (x_i^{nl}, y_i^{nl}, r_i^{nl}, w_i^{nl}, id\_fonction_i^{nl}, ...)$ portant un motif n'en utilisant pas, on s'arrange pour que tous les m_i^l soit placés en tête de liste et les m_i^{nl} en queue. On utilise alors l'algorithme suivant :

$Q_r = 0$
$\forall\ m_i^l$
    $fonction\_motif[id\_fonction_i^l](m_i^l, zone\_influence_i^l, Q_r)$
f$\forall$
$Q_r = exp(Q_r)$
$\forall\ m_i^{nl}$
    $fonction\_motif[id\_fonction_i^{nl}](m_i^{nl}, zone\_influence_i^{nl}, Q_r)$
f$\forall$

Remarque : $fonction\_motif$ est un tableau de pointeurs de fonctions, ceci évite d'utiliser un *switch* pour savoir quelle fonction appeler, ce qui serait plus lent.

1.4.2 Motif carré avec un angle de rotation quelconque

La méthode utilisée est la suivante :

- Le côté du carré de base est $\frac{1}{\sqrt{2}}$. Pour tout $m_i = (x_i, y_i, r_i, ...) \in M$, on a généré un angle θ_i uniformément dans $[0, 2\pi]$.
- Pour tout point (x, y) de la zone d'influence de m_i, on teste si (x, y) est dans le carré de la manière suivante :
  soit $\vec{u} = \begin{pmatrix} x - x_i \\ y - y_i \end{pmatrix}$
  soient $\vec{a} = \begin{pmatrix} cos\theta_i \\ sin\theta_i \end{pmatrix}$ et $\vec{b} = \begin{pmatrix} sin\theta_i \\ -cos\theta_i \end{pmatrix}$
  $(x, y) \in$ carré si $|\vec{u} \cdot \vec{a}| < \frac{r_i}{2\sqrt{2}}$ et $|\vec{u} \cdot \vec{b}| < \frac{r_i}{2\sqrt{2}}$ (voir figure 2)

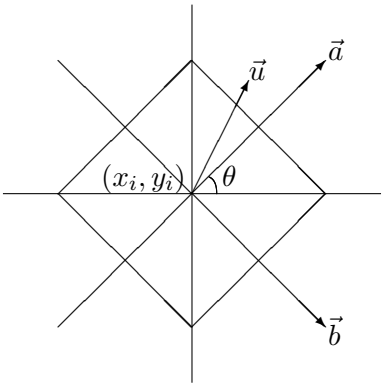

Fig. 2 – Carré avec un angle de rotation quelconque

1.4.3 Motif polygonal

On considère pour l'instant le cas où l'angle de rotation du motif est nul. Soit $S = \{s_j / \ j \in \{0, ..., n\text{-}1\}, s_j \in [-\frac{1}{2}; \frac{1}{2}]^2\}$ la suite de sommets du plan formant le polygone. Après avoir isolé une zone d'influence pour $m_i \in M$ de côté r_i, on teste l'appartenance d'un point $q_{l,m} = Q_r(l, m)$ de la zone à l'intérieur du polygone de la manière suivante :

- On trace une demi-droite partant du point $q_{l,m}$, par commodité la demi-droite $x = q_{l,m}.x$ pour $y \geq q_{l,m}.y$
- On compte combien de côtés du polygone sont intersectés $\rightarrow nb$

- Si nb est pair, $q_{l,m}$ est en dehors de la forme
- Sinon $q_{l,m}$ est à l'intérieur

Pour cela, à l'initialisation du programme, on calcule $\forall j \in \{0, ..., n\text{-}1\}, k = (j+1)[n]$

- $a_j = \frac{s_k.y - s_j.y}{s_k.x - s_j.x}$
- $b_j = s_j.y - a_j \cdot s_j.x$
- $y_{jmin} = min(y_j, y_k), y_{jmax} = max(y_j, y_k)$

Puis pour un point dans la zone d'influence d'un multiplieur m_i :

Soit S' la suite S après translation de vecteur (x_i, y_i) et homothétie de rapport r_i
Soient $x = (q_{l,m}.x - x_i)$, $y = (q_{l,m}.y - y_i)$
nb=0
$\forall j \in \{0, ..., n\text{-}1\}$
  $y_c = a_j \cdot x + b_j \cdot r_i$
  si $y \leq y_c$ et $y_c \in [y_{jmin}, y_{jmax}]$ alors
    $nb = nb+1$
  fsi
f∀
si $nb[2] == 1$ alors
  mise à jour de $Q_r(l, m)$
fsi
(voir figure 3)

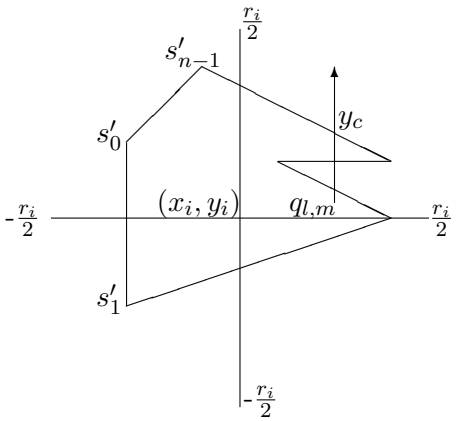

FIG. 3 – Appartenance d'un point à un polygone

1.4.4 Motif courbe spline fermée

Pour tester si un point appartient à une courbe spline, on procède de la même façon que pour les polygones. A la place d'un segment de droite, on dispose entre deux points de contrôle consécutifs de deux équations de degré 3 :

- $x_j(t), t \in [0, 1]$
- $y_j(t), t \in [0, 1]$

$j \in \{0, ..., n\text{-}1\}$
L'algorithme utilisé par calculer les splines est celui présenté dans la référence [4].
Après avoir isolé une zone d'influence pour $m_i \in M = \{(x_i, y_i, r_i, ...)/i \in \{1, .., N\}\}$ de côté r_i, on teste l'appartenance d'un point $q_{l,m} = Q_r(l, m)$ de la zone à l'intérieur de la courbe de la manière suivante :

On pose $x = (q_{l,m}.x - x_i)$, $y = (q_{l,m}.y - y_i)$.
nb=0
$\forall\ j \in \{0, ..., n\text{-}1\}$

recherche des racines de $x_j(t) - x/r_i \rightarrow s_1$ ou (s_1 et s_2) ou (s_1 et s_2 et s_3)
$\forall$ s_k tel que $s_k \in [0, 1]$
$y_c = r_i \cdot y_j(s_k)$
si $y \leq y_c$ alors
$nb = nb+1$
fsi
f$\forall$
f$\forall$
si $nb[2] == 1$ alors
mise à jour de $Q_r(l, m)$
fsi

Pour trouver les racines des polygones de degré 3, on utilise la méthode de Cardan (cf. référence [5]).

Pour faire une rotation du motif, on effectue une rotation des sommets du polygone ou des points de contrôle de la courbe spline fermée, puis on recalcule les équations des polynômes.

1.4.5 Fonction lissante particulière

Soit $(x, y) \in Q_r$ appartenant à un cercle de centre (x_i, y_i) et de diamètre r_i. La valeur de la matrice Q_r en ce point est affectée par le multiplieur $m_i = (x_i, y_i, r_i, \omega_i, ...)$ en fonction de la valeur de $r = \frac{1}{r_i} \times \sqrt{(x - x_i)^2 + (y - y_i)^2}$. Deux fonctions lissantes ont été implémentées :

- $f(r) = cos^2(\pi r)$
- $f(r) = exp(1 - \frac{1}{1-(2r)^4})$

1.4.6 Fonction lissante quelconque

Au lieu de travailler avec une fonction, on travaille avec un vecteur de points correspondant à la discrétisation de la fonction sur [0 ;0.5] avec un pas spécifié par l'utilisateur. L'approximation de la valeur de la fonction pour une abscisse donnée utilisée est l'interpolation linéaire entre les deux points de discrétisation juste avant et après l'abscisse considérée.

1.5 Décomposition d'une texture en zones

Pour les motifs de type polygone, courbe spline fermée ou ellipse, on utiliserait trop de mémoire si l'on stockait pour tout point p_i les paramètres du motif. Pour cela on utilise une variable supplémentaire, $motifs$, qui contient les paramètres de tous les motifs de ses types. Les multiplieurs m_i ne contiennent qu'un numéro qui leur permet d'accéder aux paramètres du motif qu'ils portent, dans la variable $motifs$. Ces numéros correspondent aux positions des jeux de paramètres dans la liste.

On introduit une nouvelle structure, la zone, qui correspond à une région de la texture avec sa propre loi de génération, son propre motif et sa propre densité.

Initialement, lorsque l'on génère une texture, on crée une structure qui contient :

- les limites de la zone (pour l'instant indéfinies), c'est à dire les points de contrôle de la courbe qui la délimite.
- les informations sur le motif : son type, le booléen qui indique s'il est d'angle fixe ou variable et éventuellement ses paramètres (pour le polygone, l'ellipse, ...).
- les informations sur la loi des multiplieurs, notamment leur densité.

Ensuite, lorsque l'on sélectionne une partie de la texture, on définit les limites de la zone comme les limites de la sélection. On construit ainsi après insertions et sélections successives une liste de zones. Notons qu'ayant défini une limite pour chacune des zones, on peut supprimer un zone A de la liste :

- lorsqu'en sélectionnant une région de la texture, A et la sélection sont disjoints.
- lorsqu'en insérant une région, A est entièrement masquée par la région insérée.

1.6 Sélection et insertion

On souhaite pouvoir sélectionner une partie des multiplieurs m_i ayant servi à générer une texture, puis les insérer dans une autre liste de multiplieurs d'une autre texture. Lors de l'insertion, on supprime de la liste de départ les points situés dans la zone où sont insérés les nouveaux points.

Pour créer plusieurs régions dans une texture, on considère le procédé de sélection/insertion suivant :

- on sélectionne les variables aléatoires, ayant servi à générer une texture, inscrites dans une courbe spline fermée. C'est à dire les variables dont (x_i, y_i) est inscrit dans la courbe.
- pour l'insertion d'une texture A dans une texture B, on supprime les points de B dont (x_i, y_i) est inscrit dans la courbe et on insère à la place les points de A.

Lors de la fusion des deux listes de multiplieurs, on s'arrange pour avoir toujours les points m_i^l en tête de liste et les m_i^{nl} en queue. L'indice du premier m_i^{nl} est stocké dans la variable $motifs$. Il faut, lors de l'insertion, concaténer les deux variables $motifs$. Ceci implique de modifier les numéros des jeux de paramètres dans la liste des m_i.

Contrairement à l'insertion d'une partie de la matrice Q_r (voir paragraphe 1.8), on peut zoomer dans la texture que l'on obtient. Un exemple de résultat obtenu est donné figure 4.

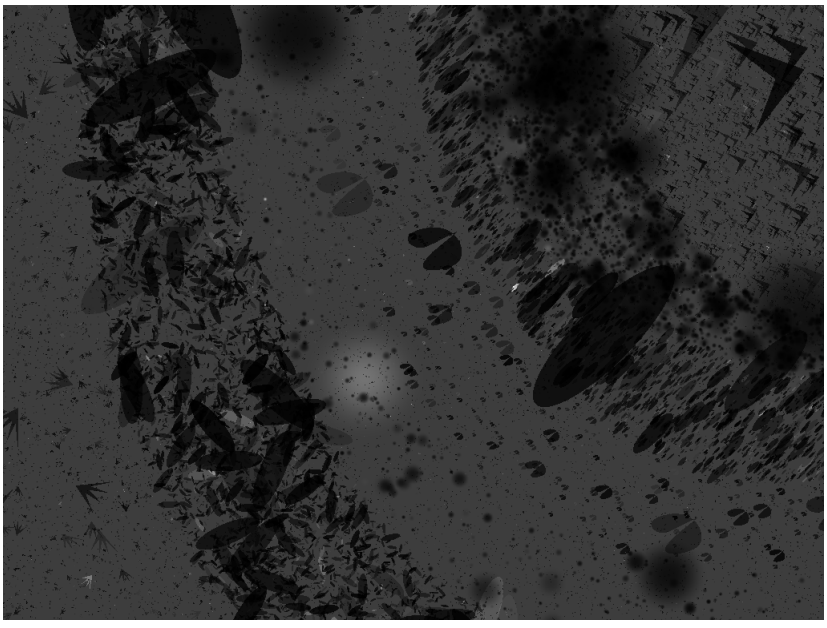

FIG. 4 – Sélections et insertions

1.7 Zoom

Soit une texture T composée de plusieurs zones $\{A, B, C, ...\}$ obtenue par le procédé de sélection et d'insertion expliqué au paragraphe précédent. On suppose que les zones sont classées

dans l'ordre ($\preceq$) où on les a insérées. Notons que si on insère une région composée de plusieurs zones $A_1 \preceq A_2 \preceq ...$ dans une texture $B_1 \preceq B_2 \preceq ...$ alors le classement est obtenu en concaténant les deux listes : $B_1 \preceq B_2 \preceq ... \preceq A_1 \preceq A_2 \preceq ....$

Initialisation : On détermine pour chaque zone le plus petit rectangle (aux côtés parallèles aux axes), restreint au cadre du zoom, dans lequel elle s'inscrit ($\rightarrow$ son cadre), comme sur la figure 5.

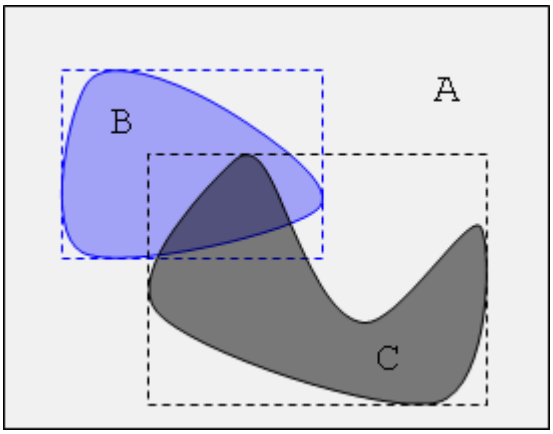

FIG. 5 – Zones de la texture

Si le cadre d'une des zones est disjoint avec le cadre du zoom, on supprime la zone.

Pour chaque zone :

- on calcule, en fonction de la densité des multiplieurs de la zone, le nombre de multiplieurs à tirer dans le cadre de la zone, on les tire.
- on supprime les multiplieurs en dehors des limites de la zone (sauf pour la première zone qui n'a pas de limites).
- on supprime les points situés en dehors du "masque" : si sur l'exemple précédent on sélectionnait une région de A incluant une partie de B et de C, alors lorsqu'on insère la sélection dans une autre texture, il faut vérifier qu'on ne génère pas de points de B ou de C en dehors des limites de A. On dit alors que A est un masque pour B et C (voir figure 6).

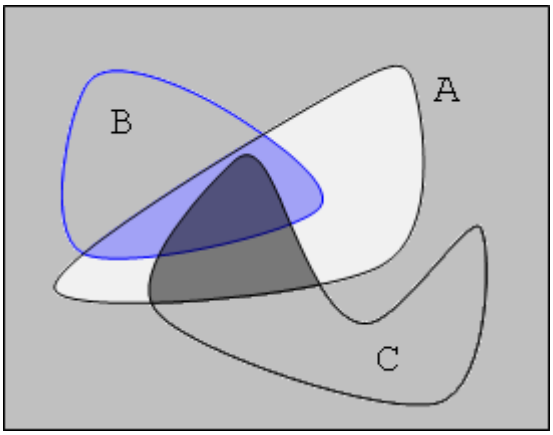

FIG. 6 – Masque de zones

- on supprime les multiplieurs situés dans une autre zone qui couvre la zone (située après la zone dans le classement).

Terminaison : Si une zone n'a plus de multiplieurs, on la supprime. Ensuite tout se passe comme dans la fonction de génération.

Remarque : on a utilisé la fonction qui détermine un cadre pour une courbe spline afin d'accélérer les fonctions de sélection et d'insertion.

1.8 Sélection d'une partie de la matrice Q_r

Pour sélectionner un partie de la matrice Q_r, on procède exactement de la même manière que pour la sélection des multiplieurs. Comme les éléments de Q_r sont ordonnés dans l'espace, on détermine le plus petit rectangle dans lequel la zone sélectionnée s'inscrit. On ne parcourt que les points de ce rectangle.

2 Interface avec Matlab

2.1 Liste des mexfiles utilisables en ligne de commandes

Fonctions implémentées en C :

- **scid2d** (**S**ynthèse de **C**ascades **I**nfiniment **D**ivisibles en **2D**) :
  Fonction qui génère une texture, ayant comme paramètres d'entrée :
  – x_{max}, y_{max} : taille de la matrice de texture a générer
  – r_{min} : résolution de la cascade
  – dx : pas d'espace
  – $choix\_loi$: 1, 2 ou 3 (voir ci-dessous)
  – $params\_loi$: paramètres de la loi
  – $choix\_loi$ ($params\_loi$)
    1. F = distribution Log-Normal (μ, σ^2,C)
    2. F = distribution exponentielle (τ,*champ inutilisé*,C)
    3. F = Dirac ($\omega_i = Cte = \omega_0$) (log-Poisson ou She-Leveque) ($\omega_0 < 1$,champ inutilisé,C)
  – motif : identifiant du motif
    1. carré
    2. cercle
    3. utilisation d'une fonction lissante ($\cos^2(\pi.r)$)
    4. utilisation d'une fonction lissante ($exp(1 - \frac{1}{1-(2r)^4})$)
    5. polygone
    6. courbe spline fermée
    7. ellipse
    8. utilisation d'une fonction lissante quelconque
  – $angle$: angle de rotation du motif (0 :fixe,1 :aléatoire)
  – $parametres\_motif$: points du polygone, points de contrôle de la spline (répéter le dernier point pour la spline) : [les x,les y] (x,y) est dans [-.5 .5]x[-.5 .5] ou [$demi\_grand\_axe$,$foyer$, $angle$] d'une ellipse ou vecteur de la fonction lissante quelconque discrétisée sur [0 ;.5] (paramètre inutilisé pour les autres motifs)
  La fonction retourne les variables suivantes :
  – Qr : matrice de texture
  – va : variables aléatoires utilisées pour la synthèse
  – $zones$: zones présentes dans la texture
  – x,y : vecteurs position $->$ $Qr(x,y)$ (facultatif)

- **zcid2d** (**Z**oom dans une **C**ascade **I**nfiniment **D**ivisible en **2D**) :
  Fonction de zoom. On utilise également la fonction pour recalculer la matrice Q_r en passant une liste de variables aléatoires modifiée par la fonction **selection_insertion** (insertion) : pour cela on effectue un zoom sur l'intégralité de la texture en ne modifiant pas la résolution.
  Cette fonction prend en entrée les arguments suivants :
  – $cadre$: cadre de la zone de la texture a agrandir [coin supérieur gauche (x et y) ;côte selon x ;côte selon y]
  – dx : pas d'espace
  – r_{min} : résolution de la cascade
  – r_{min_prec} : résolution précédente de la cascade
  – va : variables aléatoires utilisées pour la synthèse

– *zones* : zones présentes dans la texture
La fonction retourne les variables suivantes :
– Q_r : matrice de texture
– *va* : variables aléatoires utilisées pour la synthèse
– *zones* : zones présentes dans la texture
– x,y : vecteurs position $->$ $Qr(x,y)$ (facultatif)

- **selection** :
Sélection d'une partie de la matrice Q_r délimitée par une courbe spline fermée. On détermine le plus petit rectangle dans lequel s'inscrit la zone et la matrice retournée à la taille de ce rectangle. Cette fonction prend comme entrées :
– *points* : points de contrôle de la spline
– Q_r : matrice de texture
Elle retourne en sortie :
– V_r : matrice de texture extraite :

$$V_r(i-rec_i, j-rec_j) = \begin{cases} Q_r(i,j) \text{si } (i,j) \text{ est dans la zone sélectionnée ou } (rec_i, rec_j) \text{ est} \\ \quad \text{le coin supérieur gauche du rectangle} \\ 0 \text{ sinon} \end{cases}$$

- **points_spline** :
Calcul d'une liste de points pour tracer une courbe spline fermée de points de contrôle donnés. La forme de la courbe est celle fournie par l'algorithme que l'on utilise pour les motifs ou la sélection d'une partie de texture, qui diffère légèrement de la fonction *spline* de *matlab* (la dérivée de la courbe présenterait une discontinuité).
Cette fonction prend comme entrées :
– *spline* : points de contrôle de la spline (répéter le dernier point)
– *nb_points* : nombre de points a calculer entre deux points de contrôle
Elle retourne en sortie :
– *courbe* : points $[x, y]$ de la courbe calculés

- **niveaux_gris** :
Affectation à chaque point de l'image d'un niveau de gris afin qu'en zoomant, la zone de l'image agrandie conserve sa couleur. Elle réalise trois types d'actions :
– action 1 : répartition des niveaux de gris entre 0 et 255
– action 2 : répartition des niveaux de gris pour des extrema donnés
– action 3 : recherche des extrema des niveaux de gris d'une zone de l'image
Cette fonction prend comme entrées, selon le type d'actions :
– Q_r : matrice de texture
– ng_{min}, ng_{max} : extremum des niveaux de gris d'une image (si action 2)
– NG : matrice des niveaux de gris (si action 3)
– *cadre* : cadre du zoom (si action 3)
Elle retourne en sortie, en fonction de l'action à réaliser :
– NG : matrice des niveaux de gris (si action 1 ou action 2)
– ng_{min}, ng_{max} : extremum des niveaux de gris d'une image (si action 3)

- **selection_insertion** :
Sélection des variables aléatoires comprises dans une zone délimitée par une courbe spline fermée ou insertion d'une liste de variables aléatoires précédemment sélectionnée dans une autre (on supprime alors les variables présentes à l'endroit de l'insertion)
Cette fonction prend comme entrées :
– *vaC* : variables aléatoires utilisées pour la synthèse
– *zonesC* : zones présentes dans la texture courante

– vaS : variables aléatoires a insérer (insertion)
– $zonesS$: zones présentes dans la texture a insérer (insertion)
– $points$: points de contrôle de la spline (sélection)
Elle retourne en sortie :
– va : variables aléatoires utilisées pour la synthèse
– $zones$: zones présentes dans la texture courante

- **pile** :
  Gestion de l'empilement des données. Le fonctionnement de cette fonction est expliqué dans le paragraphe suivant.

2.2 Stockage des données - gestion d'une pile

On dispose d'une seule fonction pour empiler et dépiler les données dans un fichier. En fonction du nombre de ses arguments en entrée et en sortie, la fonction *pile* détermine l'action à effectuer. On a choisi de gérer la pile avec une fonction en C car avec Matlab, on ne peut pas charger une partie d'une variable. Ainsi, à chaque fois qu'on dépilerait, on devrait charger toute la pile en mémoire !

Les données que l'on stocke dans la pile sont :

- Q_r
- $va = (x_i, y_i, r_i, w_i, ...)$
- x_{max}, y_{max}
- r_{min}
- dx
- ng_{min},ng_{max}
- $zones$

On utilise un argument supplémentaire en entrée/sortie (*tailles*) qui contient la taille de chaque élément $(Q_r, va, x_{max}, y_{max}, r_{min}, dx, ng_{min}, ng_{max}, zones)$ de la pile, afin de retrouver le début et la fin des données dans le fichier.

Problème : on ne peut pas "raccourcir" un fichier, donc quand on dépile, la taille d'un fichier ne diminue pas. Cependant, lorsque la pile est vide (avant un empilement ou après un dépilement), on détruit et on recrée le fichier.

2.3 Interface graphique

2.3.1 Zoom

Lorsque l'on zoom, on recalcule un nouveau pas d'espace de telle sorte que le nombre de points tirés reste identique à celui de la texture de départ. Pour cela on prend

$$dx = dx\_prec \times \sqrt{\frac{longueur \times largeur}{x_{max} \times y_{max}}}$$

Ainsi on en déduit le nouveau r_{min}, que l'on prend proportionnel au pas d'espace :

$$r_{min} = r_{min\_prec} \times \frac{dx}{dx\_prec}$$

2.3.2 Fonction lissante quelconque

- L'utilisateur saisit une chaîne de caractères de la forme *'y=f(r) ;'* ainsi que le nombre de points de discrétisation depuis une boîte de dialogue.

- L'interface exécute le code suivant :

$$
\begin{array}{l}
answer = inputdlg(prompt, dlgTitle, lineNo, def); \\
if\ (\ isempty(answer)) \\
\quad expression = char(answer(1)); \\
\quad discretisation = str2num(char(answer(2))); \\
end \\
pas = .5/(discretisation - 1); \\
r = 0 : pas : .5; \\
eval(expression);
\end{array}
$$

- On récupère donc un vecteur y que l'on passe à $scid2d$.

2.3.3 Organisation du code de l'interface

L'interface a été réalisée à l'aide du *guide* de Matlab qui nous a généré les fichiers $texture.m$ et $texture.fig$. le *guide* a généré le squelette des fonctions qui seront appelées par Matlab, lorsque l'on clique sur un contrôle de l'interface par exemple. Ces fonctions se trouvent dans $texture.m$. Selon la taille et l'importance du code, on appelle une fonction dans un autre fichier. La répartition des fonctions dans les fichiers est donnée dans les figures 7 et 8 .

Nom de la fonction	Numéro de ligne	Description de la fonction
sauver_bmp_Callback	l. 134	Sauvegarde la texture en format bmp
sauver_eps_Callback	l. 151	Sauvegarde la texture en format eps
sauver_jpg_Callback	l. 166	Sauvegarde la texture en format jpg
quitter_Callback	l. 185	Détruit le fichier pile.dat et libère la mémoire des variables globales
sauve_donnee_Callback	l. 199	Sauvegarde toutes les données dans un fichier
Ouvrir_donnees_Callback	l. 219	Restaure toutes les données stockées dans un fichier
trois_dim_Callback	l. 290	Affiche une partie de la texture à l'aide de la fonction surf
motif_Callback	l. 443	Affiche la forme du motif dans le cas d'un polygone et d'une spline, ou la forme d'une fonction lissante (dans un petit cadre)

FIG. 7 – Emplacement des fonctions dans le fichier $texture.m$

Nom de la fonction	Nom du fichier	Description de la fonction
generer(handles)	generer.m	Récupère les paramètres de la texture et appelle le *mexfile scid2d*
zoom_avant(handles)	zoom_avant.m	Empile la texture courante, récupère le cadre saisi par l'utilisateur et appelle le *mexfile zcid2d*
zoom_arriere	zoom_arriere.m	On dépile : on récupère la texture précédente et on l'affiche
messagebox	messagebox.m	Fonction appelée à chaque fois que l'on souhaite afficher des informations sur la fenêtre de l'interface
selection1(handles)	selection1.m	Récupère les contours de la zone à sélectionner, appelle le *mexfile selection_insertion* puis sauvegarde les données (variables aléatoires) de la zone dans un fichier
insertion1(handles)	insertion1.m	Récupère dans un fichier les données de la zone à insérer, puis les insère à l'aide du *mexfile selection_insertion*
selection2(handles)	selection2(handles)	Récupère les contours de la zone à sélectionner, appelle le *mexfile selection* puis sauvegarde les données (partie de Q_r) de la zone dans un fichier
insertion2(handles)	insertion2.m	Récupère dans un fichier les données de la zone à insérer, puis les insère (en additionnant les matrices Q_r) aux endroits spécifiés par l'utilisateur
afficher(Qr)	afficher.m	Fonction qui affiche la texture soit en image soit en mapping, en utilisant le *mexfile niveaux_gris*
NGC=colorer(NG)	colorer.m	Fonction transformant une matrice indice de couleur en une matrice RGB

FIG. 8 – Répartition des fonctions dans les fichiers

Remarque : *handles* est une variable de l'interface qui permet de récupérer les paramètres saisis par l'utilisateur ou de manipuler les figures.

3 Evolution du programme

3.1 Ajouter un type de motif

3.1.1 Ajouter un motif autre qu'une fonction lissante

Pour ajouter un nouveau motif, il faut suivre la procédure suivante :

1. écrire dans *calculQr.c* une fonction de motif avec le prototype suivant :
   $void\ nom\_fonction(position\_t\ *position, tirage\_t\ *tirage)$;
   cette fonction est appelée pour chaque multiplieur. *position* contient notamment la zone d'influence du multiplieur, *tirage* contient la position, ω, θ, le booléen qui indique si l'angle du motif est constant ou aléatoire et le numéro du motif (pour accéder par exemple aux paramètres de l'ellipse, de la spline ou du polygone).
2. dans la fonction *calcul_Qr*, à la fin du fichier *calculQr.c*, rajouter le nom de la fonction dans la liste suivante :
   $void\ (*f\_motif[9])(position\_t\ *, tirage\_t\ *) = \{$
   $f\_carre\_fixe, f\_carre\_variable, f\_cercle, f\_shaper\_cos\_carre, f\_shaper\_exp,$
   $f\_polygone, f\_spline, f\_ellipse, f\_shaper\_quelconque$
   $\};$
   (sans oublier d'incrémenter le nombre d'éléments de la liste : 9)
3. dans le fichier *global.h*, rajouter un champs identifiant le motif à la fin de l'énumération $motif\_t$:
   $typedef\ enum\ \{$
   $m\_carre = 1,$
   $m\_cercle,$
   $m\_shaper\_cos\_carre,$
   $m\_shaper\_exp,$
   $m\_polygone,$
   $m\_spline,$
   $m\_ellipse,$
   $m\_shaper\_quelconque$
   $\}\ motif\_t;$
   puis un autre identifiant la fonction de motif dans l'énumération $numero\_fonction\_t$(on a deux structures car on peut avoir plusieurs fonctions pour un motif selon par exemple que l'angle du motif est fixe ou aléatoire) :
   $typedef\ enum\ \{$
   $n\_carre\_fixe,$
   $n\_carre\_variable,$
   $n\_cercle,$
   $n\_shaper\_cos\_carre,$
   $n\_shaper\_exp,$
   $n\_polygone,$
   $n\_spline,$
   $n\_ellipse,$
   $n\_shaper\_quelconque$
   $\}\ numero\_fonction\_t;$
4. dans le fichier *scid2d.c*, compléter le *switch* suivant :
   $switch\ (zone.motif)\ \{$
   $case\ m\_carre:$
   $if\ (zone.angle == fixe)\ zone.numero\_fonction = n\_carre\_fixe;$
   $else\ zone.numero\_fonction = n\_carre\_variable;\ break;$
   $case\ m\_cercle:\ zone.numero\_fonction = n\_cercle;\ break;$

```
case m_shaper_cos_carre :   zone.numero_fonction = n_shaper_cos_carre; break;
case m_shaper_exp :         zone.numero_fonction = n_shaper_exp; break;
case m_polygone :           zone.numero_fonction = n_polygone; break;
case m_spline :             zone.numero_fonction = n_spline; break;
case m_ellipse :            zone.numero_fonction = n_ellipse; break;
case m_shaper_quelconque : zone.numero_fonction = n_shaper_quelconque; break;
}
```

afin de faire correspondre le numéro de motif (et le booléen *zone.angle* par exemple) avec la fonction de motif.

5. éventuellement, écrire le prototype de la fonction de motif dans le fichier *calculQr.h*, à titre informatif.
6. dans le fichier *scid2d.c*, récupérer les arguments du motif.
   Lorsqu'on lit les paramètres du motif, on effectue le test suivant :
   `if (zone.motif < 1 || zone.motif >8) mexErrMsgTxt("Type de motif inconnu !");`
   penser à incrémenter la borne supérieure.

3.1.2 Ajouter un motif fonction lissante

Pour ajouter une fonction lissante, il faut suivre la procédure suivante :

1. suivre la même démarche que précédemment.
2. dans le fichier *global.c*, compléter la fonction *est_shaper* :

```
char est_shaper(motif_t motif) {
    return (motif == m_shaper_cos_carre || motif == m_shaper_exp
            || motif == m_shaper_quelconque);
}
```

3.2 Ajouter une loi de génération des multiplieurs

Pour ajouter une nouvelle loi de génération pour les multiplieurs, il faut suivre la procédure suivante :

1. dans le fichier *tirages.h*, ajouter un identifiant de la nouvelle loi à la fin de l'énumération *type_loi_t* :

```
typedef enum {
    log_normal = 1, log_exponential, log_poisson
} type_loi_t;
```

   puis créer une structure qui contiendra les paramètres de la loi (qui doivent être de type *double*), comme dans l'exemple suivant :

```
typedef struct {
    double mu;
    double sigma2;
} log_normal_t;
```

   puis créer une instance de cette structure dans l'union *parametres_t* :

```
typedef union {
    log_normal_t ln;
    log_exponential_t le;
    log_poisson_t lp;
} parametres_t;
```

2. dans la fonction *tirer_multiplieurs* écrite dans le fichier *tirages.c*, ajouter le code pour générer la valeur des ω_i :

```
switch (loi.type) {
    case log_normal :
        cste = sqrt(p.ln.sigma2);
        for (i = 0; i < positions.n; i++) tirages[i].w = exp(cste * randn() + p.ln.mu);
        break;
    case log_exponential :
        cste = pow(1. + p.le.Temp, 1./p.le.Temp);
        for (i = 0; i < positions.n; i++) tirages[i].w = pow(cste * randu(), p.le.Temp);
        break;
    case log_poisson :
        for (i = 0; i < positions.n; i++) tirages[i].w = p.lp.W0;
        break;
}
```

3. dans le fichier *scid2d.c*, récupérer les arguments de la loi et calculer les variables $H1$ et $Ttheo2$ pour la validation des paramètres et la normalisation de Q_r.
   Lorsqu'on lit les paramètres de la loi, on effectue le test suivant :
   `if (zone.loi.type < 1 || zone.loi.type >3) mexErrMsgTxt("Loi inconnue !");`
   penser à incrémenter la borne supérieure.

Conclusion

Les principaux objectifs fixés ont été atteints : disposer d'un ensemble de fonctions rapides avec une interface graphique réalisant les fonctionnalités majeures souhaitées. Nous n'avons pas eu suffisamment de temps pour réaliser toutes les fonctions de motif envisagées ainsi que diverses autres options d'importance secondaire. Cependant, à l'aide des indications données dans ce rapport, il est possible de faire évoluer assez simplement le programme.

Nous n'avons pas rencontré de problèmes d'ordre théorique particuliers au cours de ce projet, une fois compris le principe de la synthèse. Les difficultés rencontrées ont plutôt été d'ordre pratique, que ce soit dans l'optimisation des temps de calcul et de l'utilisation de la mémoire ou dans la réalisation de l'interface graphique. La multiplicité des cas d'utilisations à prévoir lors de l'utilisation combinée des différentes fonctionnalités du programme a également été source de difficultés.

Références bibliographiques

[1] http ://www.math.sci.hiroshima-u.ac.jp/∼m-mat/eindex.html
M. MATSUMOTO, T. NISHIMURA, Mersenne Twister : A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. ACM Transactions on Modeling and Computer Simulation, vol. 8, no. 1, pp. 3-30, 1998

[2] http ://www.agner.org/random/

[3] http ://www.taygeta.com/random/gaussian.html
BOX, G.E.P, M.E. MULLER, A note on the generation of random normal deviates, Annals Math. Stat, V. 29, pp. 610-611, 1958

[4] http ://www.mathworks.com/moler/interp.pdf

[5] R.W.D. NICKALLS, A new approach to solving the cubic : Cardan's solution revealed, The Mathematical Gazette, vol. 77, 1993