Nantes Université

L3 Informatique
parcours Maths-informatique

Projet ASD3

Art abstrait et arbres : le projet Mondrian

Etudiants :
Elizabeth Gandibleux et Valentin Guy-Deroubaix

Groupe : 581

Enseignante :
Mme Irena Rusu

Résumé Piet Mondrian est un peintre néerlandais du début du 20ème siècle, pionnier de l'art abstrait. Son style est caractérisé par des oeuvres très géométriques. Le but de ce projet est d'utiliser des structures de données vues en cours pour générer des images à la manière des tableaux de Mondrian. Deux réalisation informatiques sont présentées dans ce rapport. La première, nommée *méthode1* suit le cahier des charges décrit dans le sujet du projet. Elle met en œuvre un KD-tree et un AVL-tree. La seconde, *méthode 2*, apporte une réponse à une des limitations implicites à la structure de données utilisée dans la méthode 1, et permet de générer des images proches du tableau « *Composition with Red, Black, Blue and Yellow (1928)* » de Mondrian. Elle repose sur un Quad-tree.

1 Vue macro de la méthode 1 sous forme de pseudo code

L'algorithme 1 donne une vue macroscopique des informations et traitements intervenants dans la méthode 1 (generateRandomTree). Un KD-tree (2D-tree) et un AVL-tree sont mis en œuvre. Le KD-tree permet de représenter l'image et ses informations décrivant un tableau. L'AVL-tree est utilisé pour accéder efficacement aux informations recherchées. Les noeuds de l'AVL ont pour valeur un pointeur vers les noeuds du KD-tree.

Algorithm 1 generateRandomTree : retourne un Mondrian
`public kdTree generateRandomTree(kdTree A, AVL B, int nbleaf)`

• Principaux paramètres utilisés dans le traitement :
- entier hauteur, largeur // hauteur et largeur (en nombre de pixels) de l'image générée
- entier nbFeuilles // nombre de feuilles maximum de l'arbre
- entier minDimensionCoupe // taille minimum des dimensions d'une région pour pouvoir la diviser
- réel memeCouleurProb // probabilité de garder la couleur du parent lors d'une division
- entier largeurLigne // largeur (en pixels) de la ligne qui sépare les régions
- entier proportionCoupe // valeur qui permet de ne pas découper trop proche des bords de la région
- entier seed // graine aléatoire utilisée pour générer l'arbre

• Principales structures construites au cours du traitement :
- KD-tree représentant l'image d'un Mondrian
- AVL-tree travaillant sur le poids d'un élément du KD-tree

• Principales informations en sortie du traitement :
- KD-tree représentant l'image d'un Mondrian

• Principales étapes :

1. Vérification préliminaire :
   Le nombre de feuille est-il positif ? (test sur la valeur fournie en entrée)

2. Traitement de la racine :
   Initialiser avec (1) coordonnées, (2) longueur et largeur, (3) couleur choisie aléatoirement

3. Division : `public void chooseDivision(kdTree A)`
   (a) Choix aléatoire de l'axe X ou Y de découpe (règle probabiliste donnée)
   (b) Etablir les limites de coupe selon les valeurs de paramètres (règle de proportion donnée)
   (c) Choix aléatoire des coordonnées de coupe dans les limites autorisées ainsi établies
   ↳ On dispose d'un fils gauche et d'un fils droit suite à la division

4. Insertion dans kd-tree : `public void insertion(boolean b)`
   (a) Insertion du fils gauche et lui attribuer une couleur choisie aléatoirement selon la probabilité du parent
   (b) Insertion du fils droit et lui attribuer une couleur choisie aléatoirement selon la probabilité du parent

5. Insertion dans AVL : `public int insertValueAVL(kdTree x)`
   (a) Insertion du fils gauche si conditions (minDimCoupe) respectées
   (b) Insertion du fils droit si conditions (minDimCoupe) respectées

6. Recherche dans AVL la feuille plus importante (plus gros poids) `public kdTree chooseLeaf(AVL B)`
   (a) Appel a chooseLeaf qui utilise la fonction max dans AVL
   (b) Obtient C (pointe vers un kd-tree), depuis la feuille identifiée dans l'AVL
   ↳ On dispose de C, le « plus gros fils » qui va être coupé

7. Poursuite : selon que C
   (a) Existe (non nul) ∘ Vérifier si il existe plus de 1 feuille. Si Vrai alors C est interne
   ∘ Appel à `generateRandomTree(C,B,nbleaf-1)`
   (b) sinon Retourner A (racine du kd-tree)

---

2 Description des méthodes attendues

2.1 ChooseLeaf

2.1.1 Pseudo-code

Algorithm 2 ChooseLeaf : retourne la feuille qui a le plus gros poids

```
Require: AVL B
  if estVide(B) then
    retourner null
  else
    kdTree leafBiggest ← max(B)
    enlever(B, leafBiggest)
    retourner leafBiggest
  end if
```

2.1.2 Explications du fonctionnement de la procédure

On passe en paramètre l'AVL qui contient les noeuds du kdTree généré par **generateRandomTree**. Dans un premier temps, on vérifie que l'AVL B n'est pas vide, si c'est le cas on retourne null. Dans un second temps, l'appel à la fonction max, nous sert à récupérer le plus gros poids dans notre AVL. Ce dernier est référencé par notre pointeur **leafBiggest**. Une fois utilisé, on sait que l'on ne se servira plus du plus gros noeud. De ce fait, il sera supprimé. Pour finir, on retourne **leafBiggest**.

2.1.3 Complexité de la procédure

Comme la fonction se repose sur la fonction max et enlever qui sont toutes les deux dans le pire cas, en $O(hauteur(B))$, la fonction a au pire cette complexité. Au mieux la complexité est en $\Omega(1)$.

2.2 chooseDivision

2.2.1 Pseudo-code

Algorithm 3 chooseDivision : retourne l'axe et les coordonées de la division d'une feuille donnée

```
Require: kdTree A ;

  valeurAléatoire ← random(A) ;
  setter(A) ← (valeurAléatoire ≤ poids(A)/(longueur(A)+largeur(A)))

  if axe(A) = vrai then
    limiteDeCoupe ← longueur(A) ;
  else
    limiteDeCoupe ← largeur(A) ;
  end if

  limiteMaxDeCoupe ← (limiteDeCoupe * (1 - proportionDeCoupe)) ;
  limiteMinDeCoupe ← (limiteDeCoupe * proportionDeCoupe) ;
  coordonnéesDeLaCoupe ← (random(A)(limiteMaxDeCoupe, limiteMinDeCoupe)+ limiteMinDeCoupe) ;

  if getterAxe(A) = vrai then
    coordonnéesDeLaCoupe ← coordonnéesDeLaCoupe + getterPointSuperieurGauche(A).getterX(A) ;
  else
    coordonnéesDeLaCoupe ← coordonnéesDeLaCoupe + getterPointSuperieurGauche(A).getterY(A) ;
  end if

  setterDivision(A) ← coordonnéesDeLaCoupe ;
```

2.2.2 Explications du fonctionnement de la procédure

La fonction **chooseDivision** prend pour paramètre le KdTree A. En premier, nous allons déterminer l'axe de coupe. Pour cela, on va tirer un nombre au hasard **valeurAléatoire**, puis la comparer au poids de la feuille **A** divisé par sa longueur additionnée à sa largeur. Suivant cela, **axe(A)** sera vrai ou faux. Si il est vrai, on va couper en longueur, sinon en largeur. Par la suite, on va calculer les limites de coupe, i.e. les valeurs qui nous permettent de ne pas couper trop proche du bord de la feuille. On calcule ensuite les coordonnées de la coupe. Pour finir, si **getterAxe(A)** est vrai, sachant que on ne le modifie pas donc si il est vrai avant, il est encore vrai, la **coordonnéesDeLaCoupe** va prendre une valeur qui nous permettra de couper précisément sur l'axe des X. Pour faire cette dernière action, le setter de division de A va prendre comme valeur la coordonnée de la coupe.

2.2.3 Complexité de la procédure

chooseDivision est une méthode qui fait appel à des fonctions toutes dans le pire cas constantes. Elle est donc en $\Theta(1)$, $\Omega(1)$ et $O(1)$.

2.3 chooseColor

2.3.1 Pseudo-code

Algorithm 4 chooseColor : retourne la couleur de la feuille créée

```
Require: kdTree parent, kdTree enfant ;
  if estRacine(enfant) then
    selectionCouleur(enfant, listeCouleur[aléatoireEntier(0,5)])
  else
    if aléatoireRéel(0,1) ≤ memCouleurProba then
      selectionCouleur(enfant,parent.couleur)
    else
      selectionCouleur(enfant,listeCouleur[aléatoireEntier(0,5)])
    end if
  end if
```

2.3.2 Explications du fonctionnement de la procédure

On passe en paramètre les KdTree parent et enfant, dans le but de déterminer la couleur du noeud enfant suivant celle du parent. Tout d'abord on vérifie que l'enfant n'est pas la racine de l'arbre principal dans le code. Si il l'est, alors on lui calcule une couleur aléatoirement. Ensuite, on compare le nombre aléatoire tiré au paramètre **memeCouleurProba**. Si ce dernier est supérieur ou égal au nombre tiré plus tôt, alors l'enfant prend la couleur de son parent, sinon on lui calcule une couleur aléatoirement.

2.3.3 Complexité de la procédure

Toute les fonctions utilisées dans **chooseColor** sont constantes. Ainsi, dans le pire des cas, la fonction sera en $O(1)$ et en général en $\Theta(1)$.

2.4 generateRandomTree

2.4.1 Pseudo-code :

Algorithm 5 generateRandomTree : retourne un arbre kdTree aléatoirement créée

```
Require: AVL B, kdTree A, entier nbFeuilles ;
  if nbFeuilles ≤ 0 then
    retourner A ;
  else
    if getterIsRoot(A) = vrai then
      nbFeuilles = nbFeuilles -1 ;
      if nbFeuilles ≤ 0 then
        setterIsExtern(A) ← false ;
      end if
      getterPointSuperieurGauche(A).setterXY(A) ← (0,0) ;
      setterLongueur(A) ← this.width ;
      setterLargeur(A) ← this.height ;
      chooseColor(this) ← (A,A) ;
    end if

    chooseDivision(this) ← (A) ;
    insertion(A) ← true ;
    chooseColor(this) ← (A, getterFilsGauche(A)) ;
    insertion(A) ← faux ;
    chooseColor(this) ← (A, getterFilsDroit(A)) ;

    if getterFilsGauche(A).getterLongueur(A) > minDimensionCoupe(this)
        et getterFilsGauche(A).getterLargeur(A) > minDimensionCoupe(this) then
      insertion(B) ← getterFilsGauche(A) ;
    end if
    if getterFilsDroit(A).getterLongueur(A) > minDimensionCoupe(this)
        et getterFilsDroit(A).getterLargeur(A) > minDimensionCoupe(this)  then
      insertion(B) ← getterFilsDroit(A) ;
    end if

    this.C ← this.chooseLeaf(B) ;
    if  C ≠ null then
      if  nbFeuilles > 1 then
        setterIsExtern(C) ← false ;
      end if
      retourner generateRandomTree(C,B,nbleaf-1) ;
    else
      retourner A ;
    end if
  end if
```

2.4.2 Explications du fonctionnement de la procédure

On passe en paramètre le **2D arbre A**, sur lequel on va travailler, son **AVL B** et **nbLeaf**, le nombre de feuilles voulues par l'utilisateur. Dans un premier temps, on vérifie que le nombre de feuilles, **nbLeaf**, donné par l'utilisateur est bien strictement positif, sinon on retourne l'arbre. Par la suite, on va construire le **2D-arbre**, en commençant par la racine. On commence par vérifier si l'utilisateur souhaite plus de 1 feuille. Si c'est le cas, on en conclut que notre racine n'est pas un noeud externe, sinon la racine est une feuille. Ensuite, on va calculer l'axe et les coordonnées de division, pour le 2D arbre **A**, puis créer ses deux fils et les insérer dans l'arbre. Avant de finir, on va ajouter les deux fils qui viennent d'être créé dans l'AVL, en s'assurant que les dimensions soient respectées. Pour finir, on va refaire ce schéma mais cette fois la racine sera remplacée par la plus grosse feuille **C** qui deviendra un noeud interne ; si on a bien un nombre de feuille plus grand que 1 et que **C** n'est pas une feuille (noeud externe).

2.4.3 Complexité de la procédure

La fonction **generateRandomTree** fait appel à un arbre contenant au pire **nbFeuilles** éléments, mais aussi à la fonction **chooseLeaf** qui est en O(hauteur(B)) à chacun de ses appels récursifs (B étant l'AVL). Ainsi, **generateRandomTree** est au pire en O(**nbFeuilles** × hauteur(B)) et au mieux elle ne comporte que un éléments, donc en $\Omega(1)$.

2.5 toImage

2.5.1 Pseudo-code :

Algorithm 6 toImage : retourne une image (à partir d'un arbre)

```
Require: Image img, kdTree tree ;
  pointSupLeftX ← getterPointSuperieurGauche(tree).getterX(tree) ;
  pointSupLeftY ← getterPointSuperieurGauche(tree).getterY(tree) ;
  additionLongueurLignes ← largeurLigne/2 ;
  if getterIsExtern(tree) ≠ null then
    toImage(img, getterFilsGauche(tree)) ;
    toImage(img, getterFilsDroit(tree)) ;
    if getterAxe(tree) = vrai then
      xmin = getterDivision(tree) - additionLongueurLignes ;
      xmax = getterDivision(tree) + additionLongueurLignes ;
      ymin = pointSupLeftY ;
      ymax = pointSupLeftY + getterLargeur(tree) ;
      setRectangle(img) ← (xmin,xmax,ymin,ymax,Color.GRAY) ;
    else
      xmin = pointSupLeftX ;
      xmax = pointSupLeftX + getterLongueur(tree) ;
      ymin = getterDivision(tree)-additionLongueurLignes ;
      ymax = getterDivision(tree)+additionLongueurLignes ;
      setRectangle(img) ← (xmin,xmax,ymin,ymax,Color.GRAY) ;
    else
      xmin = pointSupLeftX ;
      xmax = pointSupLeftX + getterLongueur(tree) ;
      ymin = pointSupLeftY ;
      ymax = pointSupLeftX + getterLargeur(tree) ;
      setRectangle(img) ← (xmin,xmax,ymin,ymax,getterCouleur(tree)) ;
      retourner img ;
    end if
    retourner img ;
  end if
```

2.5.2 Explications du fonctionnement de la procédure

toImage va prendre en paramètres une **Image canvas** et le **2D arbre** nommé **tree** ici. Pour faciliter les calculs, on déclare trois variables, **pointSupLeftX** qui est le point supérieur gauche de notre image sur l'axe des X, **pointSupLeftY** comme le précédent mais sur l'axe des Y et **addWidthLine** qui est la moitié de l'épaisseur de la ligne grise que l'on va venir ajouter sur l'image canvas. Pour commencer, on vérifie que notre arbre n'est pas une feuille, i.e. noeud externe. Si c'est le cas, on retourne l'image **canvas** résultante. Sinon on se déplace dans l'arbre **tree** pour traiter notre image. Si le noeud que l'on traite a son **getterAxe(tree)** qui est vrai, alors ça signifie que on va créeer une ligne grise qui sera verticale, elle sera sinon horizontale sur l'image traitée résulante.

2.5.3 Complexité de la procédure

La fonction **toImage** fait appel à la méthode **setRectangle** qui est en $O(n \times m)$ où n est la longeur de la zone à colorier et m sa largeur. Or, comme on parcours tous les éléments de la zone, on obtient donc une complexité au pire en $O(\textbf{nbFeuilles} \times n \times m\ )$ et au mieux en $\Omega(n \times m)$.

2.6 generateBetterRandomTree

2.6.1 Regards sur les résultats de la méthode 1 et exposé des choix de la méthode 2

Les résultats obtenus avec la méthode 1 sont présentés pour différentes valeurs des paramètres en section 4.1, figures 2 à 5. La méthode de construction par KD-tree et les paramètres intégrés ne permettent pas de produire des images comportant des symétries, ni d'introduire une certaine modération dans le coloriage, par exemple apportée par le blanc.

La méthode 1 ne permet donc pas de reproduire plusieurs tableaux de l'artiste [1] dont le ***Composition with Red, Black, Blue and Yellow (1928)*** illustré à la figure 1. Cette œuvre est remarquable pour (i) l'aspect symétrique/régulier de ses découpes en blocs, (ii) la dominance de rectangles blancs, (iii) le nombre de niveaux de découpe limité, (iv) la limitation à 3 rectangles de couleurs vives (rouge, jaune, bleu) sur l'ensemble du tableau, (v) l'opposition entre rectangles quasi carrés et rectangles étroits.

FIGURE 1 – Le tableau originel du peintre sur lequel on s'appuye pour notre 2 ème méthode (image tirée du site : *https ://www.centrepompidou.fr/fr/ressources/oeuvre/cpnn4xq*)

Etant donné ces observations, nous faisons le choix d'un Quad-tree (QT) pour la méthode 2 (generateBetterRandomTree), structure qui introduit naturellement la symétrique souhaitée dans la décomposition via les 4 quadrants. La méthode a un paramètre `depthMaxQT`, fixant la hauteur maximale de l'arbre.

1. Le papier de Martin Skrodzki et Konrad Polthier, *Mondrian Revisited : A Peek into the Third Dimension* de 2018 aborde le sujet en présentant deux tableaux et propose une méthode pour le premier tableau. Nous avons choisi de travailler sur le second tableau mentionné dans ce papier et pour lequel les auteurs ne proposent pas de méthode d'amélioration.

2.6.2 Pseudo-code :

Algorithm 7 generateBetterRandomTree : construit un arbre QT suivant nos choix définis pour la méthode 2

Require: PQuadTree A, int profondeur ;

```
PQuadTree ptr ;
val ← tableau d'entiers représentant une permutation aléatoire des nombres 1,2,3

while profondeur < profondeurMax do
  // (x1,y1), (x2,y2) : coordonnées du rectangle à découper
  x1 ← ptr(x1) ; x2 ← ptr(x2) ;   y1 ← ptr(y1) ; y2 ← ptr(y2) ;

  if profondeur ≠ 2 then
    arrondi ← un double aléatoire entre 0.0 et 0.2 ;
  else
    arrondi ← un double aléatoire entre 0.0 et 0.4 ;
  end if

  xm ← (arrondi × ((x2 − x1)/2) + (x2 − x1)/2 + x1) ;
  ym ← (arrondi × ((y2 − y1)/2) + (y2 − y1)/2 + y1) ;

  if profondeur = 0 then
    col ← getterCouleur3(val[0]) ;
  else if profondeur = 1 then
    col ← getterCouleur3(val[1]) ;
  else if profondeur = 2 then
    col ← getterCouleur3(val[2]) ;
  end if

  quadrant ← un entier aléatoire entre 1 et 4 ;
  if quadrant = 1 then
    col1 ← col ; col2 ← blanc ; col3 ← noir ; numNoeudInterne ← 3 ; col4 ← blanc ;
  else if quadrant = 2 then
    col1 ← blanc ; col2 ← col ; col3 ← blanc ; col4 ← noir ; numNoeudInterne ← 4 ;
  else if quadrant = 3 then
    col1 ← noir ; numNoeudInterne ← 1 ; col2 ← blanc ; col3 ← col ; col4 ← blanc ;
  else if quadrant = 4 then
    col1 ← blanc ; col2 ← noir ; numNoeudInterne ← 2 ; col3 ← blanc ; col4 ← col ;
  end if

  // Appel à notre fonction d'insertion dans le QT qui retourne un pointeur sur le noeud
  // candidat à être découpé au niveau suivant
  ptn ← ptr.ajouter4NoeudsExternes(ptr, x1, x2, y1, y2, xm, ym, col1, col2, col3, col4, numNoeudInterne) ;

end while

retourner A ;
```

2.6.3 Explications du fonctionnement de la procédure

La fonction **generateBetterRandomTree** prend en paramètre un **QuadTree** A et un entier qui indique le niveau de profondeur courant lors de la création du quad-tree.

Le QT est construit niveau par niveau avec les particularités suivantes :
— La sobriété des couleurs est obtenue en coloriant complètement en rouge/bleu/jaune un seul fils par niveau ;
— La fluidité apportée par le blanc est obtenue en coloriant en blanc sur le niveau les deux quadrants adjacents au quadrant colorié ;
— Les noeuds coloriés forment des noeuds externes du QT ;
— Enfin le dernier quadrant restant (non colorié, noeud interne dans le QT) sera découpé.

Lorsque l'on regarde le tableau du peintre, on observe qu'il y a trois couleurs vives (jaune, rouge et bleu) et deux couleurs neutres (blanc et noir). Aussi, chaque couleur parmi rouge/bleu/jaune n'apparaît qu'une seule fois (une couleur par découpe).

Ceci est mis en place dans notre code en générant une permutation aléatoire des chiffres 1/2/3. Pour cela, un tableau allant de 1 à 3 est déclaré, et il prend aléatoirement une de ces 3 couleurs vives une et une seule fois. Ensuite, tant que profondeur n'a pas atteint la profondeur maximale, on colorie/découpe un quadrant en 4 quadrants.

Les valeurs pour réaliser une découpe sur les dimensions X et Y sont choisies autour du point milieu avec un facteur aléatoire pour générer des rectangles quasi carrés à la première découpe, et des rectangles plus étroits lors de la découpe suivante. Pour se rapprocher de l'esthétique de l'œuvre, au plus 3 niveaux de découpe seront réalisés (bien que notre méthode peut aller au delà de 3, comme nous le montrons dans les résultats avec les figures 8 et 9).

Par la suite, on va devoir choisir quel quadrant colorier ou découper. On prend donc un nombre aléatoire entre 1 et 4 qui va correspondre à la place des quadrants : 1 celui en haut à gauche (NO), 2 en haut à droite (NE), 3 en bas à droite (SE) et 4 en bas à gauche (SO). Suivant le résultat, la partie choisie sera coloriée soit en *rouge, jaune* ou *bleu*, ses côtés adjacents seront blancs et le quadrant opposé devra être découpé.

On répète ce processus jusqu'à arriver au niveau final où la feuille sensée être découpée vers un niveau inférieur, mais qui ne le sera pas du fait que le nombre de niveau maximum est atteint (paramètre `depthMaxQT` atteint), sera coloriée en noir, qui est la couleur par défaut choisie.

La classe qui comporte les traitements relatifs au quad-tree se réduit à des constructeurs, des getter/setter et une méthode d'insertion dans le quad-tree. Comme on dispose en permanence d'un pointeur sur l'élément à décomposer (nœud interne), l'insertion dans le QT se fait immédiatement. La décomposition et l' arbre obtenu se présentent par exemple de la manière suivante (le couple ^^ représente un pointeur vers le niveau suivant) :

```
box: 0 0 1000 1000     | cut: 411 411
Quadrant 1 (NO) colorié et 3 (SE) décomposé
box: 411 411 1000 1000 | cut: 697 697
Quadrant 2 (NE) colorié et 4 (SO) décomposé
box: 411 697 697 1000  | cut: 566 861
Quadrant 3 (SE) colorié et 1 (NO) décomposé

Level : 1  ----------
NO NE ^^ SO
SE interne
Level : 2  ----------
NO NE SE ^^
SO interne
Level : 3  ----------
^^ NE SE SO
NO interne -> externe et colorié en noir
```

2.6.4 Complexité de la procédure

La fonction **generateBetterRandomTree** est au pire linéaire et au mieux constante.

3 Commandes de compilation et d'exécution en mode console

Le cheminement est le suivant : (1) ouvrir un terminal, (2) se déplacer dans le répertoire qui contient tous les fichiers du projet et (3) entrer à l'invite de commandes les deux lignes suivantes :

```
javac *.java        commande pour compiler le projet
java Main           commande pour exécuter le programme
```

Cette seconde commande exécute le programme en prenant les valeurs des paramètres établis par défaut dans le code source. La table 1 décrit les paramètres et l'ordre dans lequel ils sont pris en compte (les paramètres communs aux 2 méthodes en haut de la table, les spécifiques à méthode 1 au centre, et les spécifiques à méthode 2 en bas).

type	nom du paramètre	valeur par défaut	commentaire
entier	strategy	1	méthode (1 ou 2) utilisée
entier	largeur	1000	largeur (pixels) de l'image
entier	hauteur	1000	hauteur (pixels) de l'image
entier	largeurLigne	5	largeur (pixels) trait séparation
booléen	seed	0 (= seed aléatoire)	si seed > 0 : séquence random reproductible
entier	nbFeuilles	4000 (valeur max)	nombre de feuilles
entier	minDimensionCoupe	5	taille min des dim / région pour diviser
flottant	memCouleurProb	0.3	prob. garder couleur du parent lors division
flottant	proportionCoupe	0.2	pas découper proche des bords région
entier	depthMaxQT	3	profondeur du Quad-tree

Table 1 – Paramètres du programme et leurs informations

L'utilisateur peut exécuter le programme avec ses propres paramètres. Il doit les renseigner en donnant nécessairement tous les paramètres attendus (le nombre et l'ordre des paramètres doivent être strictement respectés). Les deux commandes suivantes illustrent l'exécution du programme sur l'une ou l'autre méthode :

- `java Main 1 500 500 5 0 20 5 0.3 0.2` (exécuter le programme sur la méthode 1)
- `java Main 2 1000 1000 5 0 3` (exécuter le programme sur la méthode 2)

4 Exemples de résultats obtenus et discussion

Pour ces exemples, les valeurs données en légende des figures 2 à 7 correspondent respectivement aux paramètres de la table 1.

4.1 Méthode n°1

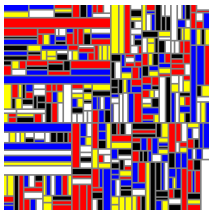

FIGURE 2 – (1 1000 1000 8 10000 450 4 0.01 0.3)

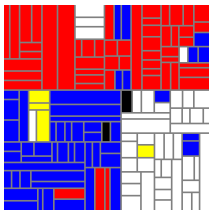

FIGURE 3 – (1 1000 1000 8 30000 150 4 0.9 0.3)

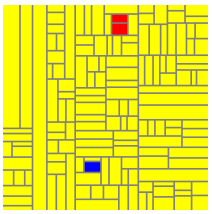

FIGURE 4 – (1 1000 1000 8 808 150 4 0.99 0.3)

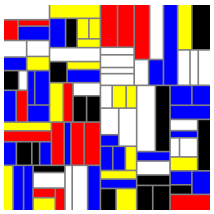

FIGURE 5 – (1 1000 1000 8 2501 100 4 0.5 0.3)

4.2 Méthode n°2

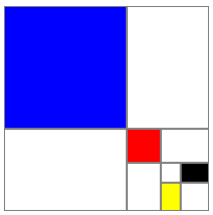

FIGURE 6 – (2 1000 1000 5 25 3)

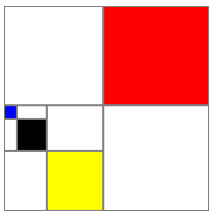

FIGURE 7 – (2 1000 1000 5 15 3)

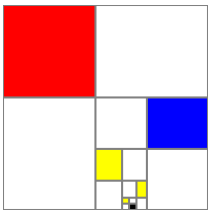

FIGURE 8 – Profondeur de 5

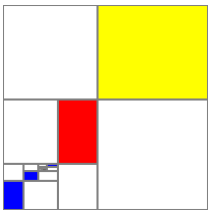

FIGURE 9 – Profondeur de 6

4.3 Discussion et remarques

Touts les codes ont été programmés en Java. La solution informatique qui implémente la méthode 1 est opérationnelle et répond aux attentes lorsque l'utilisateur fait varier les paramètres de génération d'une image. Bien que nous avons été attentifs aux complexités des algorithmes mis en oeuvre, la méthode 1 rencontre une limite technique et lève un *stack overflow* autour de nbFeuilles=4500 feuilles. Le paramètre par défaut est donc fixé à 4000 feuilles. Nous avons remarqué que cette limite s'est resserrée en accédant aux attributs des classes via getter et setter. Nous avons fait un test rapide qui a consisté à accèder aux attributs en référence directe, le problème n'a plus été rencontré pour 5000 feuilles. Nous supposons que l'accès recommandé en programmation orientée objet aux objets par getter/setter provoque une occupation mémoire plus grande qui, du fait de la nature récursive de nos codes, sature le stack alloué à un programme en exécution.

La solution informatique que nous avons développé qui implémente la méthode 2 permet de générer des images (figures 6 et 7) qui comportent les principales caractéristiques attendues du tableau ciblé de Mondrian. Des améliorations peuvent être apportée à notre génération notamment en favorisant davantage la présence de rectangles blancs en zone centrale de l'image et donc mettre en couleur les rectangles de petites tailles adjacentes au bord du tableau. Ceci est envisageable en ajoutant à notre procédure de coloriage des règles qui vont dans ce sens. Aussi, notre solution informatique permet l'aller au delà des 3 niveaux de découpes. Mais au delà de 3 niveaux, le résultat s'éloigne des caractéristiques que l'artiste a mis dans son tableau (figures 8 et 9). Par ailleurs, une condition d'arrêt basée sur la taille du rectangle courant, intégrant ou pas l'épaisseur des traits de bordure du rectangle, serait à prévoir afin de stopper la décomposition avant d'avoir atteint la profondeur souhaitée.