Manipulation d'images représentées récursivement

Frédéric DURAND et François FLEURET
Sujet proposé par François BOULADOUX
E-mail : `fdurand@clipper.ens.fr`, `fleuret@clipper.ens.fr`

24 avril 1994

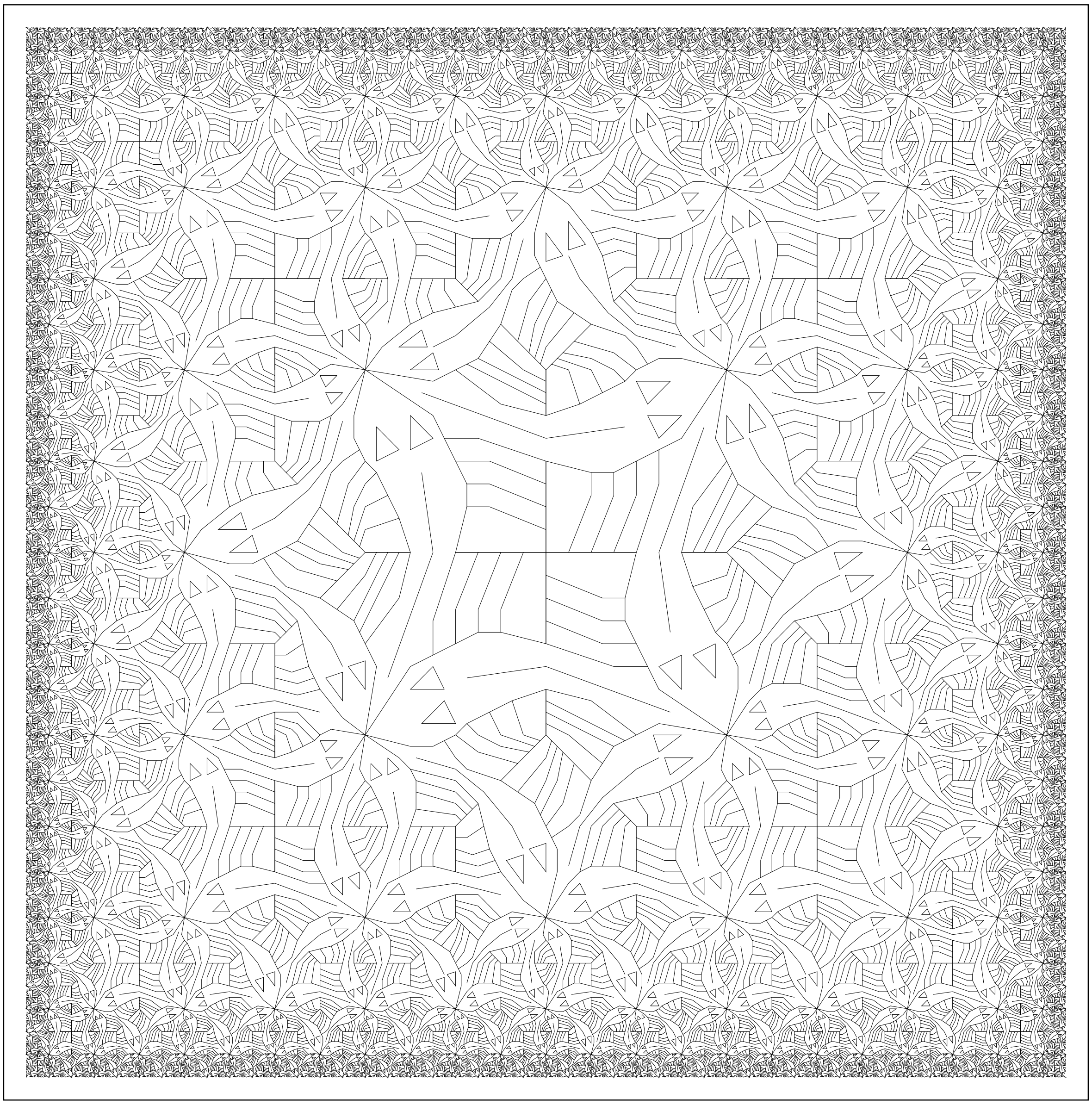


FIG. 1 - Limite carrée *par M.C.* ESCHERS

1 Représentation d'image avec des arbres quaternaires

Une méthode classique utilisée pour représenter des images carrées consiste à définir une image comme étant soit une image de couleur uniforme, soit la réunion de quatre sous-images carrées de meme tailles. Ainsi, toute une catégorie d'images peut être représentée sous la forme d'arbres quaternaires dont les feuilles sont des couleurs. Dans toute la suite, sauf précision contraire, on se limitera à deux couleurs : le noir et le blanc, et on pourra ainsi identifier une image avec une partie du carré élémentaire (L'image toute blanche est l'ensemble vide, etc.). De plus, on numérotera les fils d'un nœud dans le sens des aiguilles d'une montre, en partant du quart nord-ouest. Par exemple :

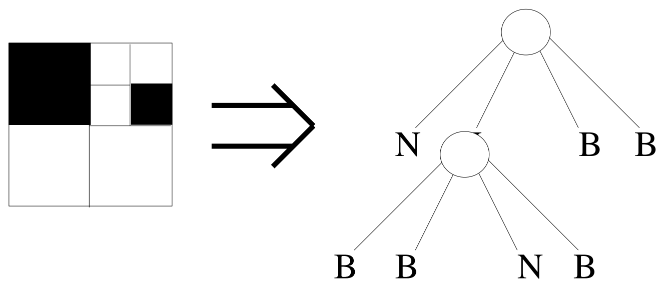


FIG. 2 - *Un exemple de quad-tree*

Cette méthode de représentation permet de minimiser l'information nécessaire pour coder des zones homogènes ; de ce point de vue, c'est une forme de compactage.

Néanmoins, l'ensemble des images qui peuvent etre représentées par cette méthode est assez limité, en effet, on voit facilement que les seules images qui peuvent etre représentées par des arbres quaternaires (finis) sont des réunions finies de carrés dont les sommets ont des coordonnées de la forme $\frac{p}{2^k}$.

Dans ce projet on se propose d'étendre cette représentation en y introduisant la récursivité. Un nœd pourra donc possèder pour fils des images situées au dessus de lui dans l'arbre. Par exemple considérons l'image :

$$X = \begin{pmatrix} Blanc & X \\ X & Noir \end{pmatrix}$$

Elle représente un triangle inférieur droit.

Cette représentation est beaucoup plus riche que la précédente qu'elle contient trivialement (il suffit de considérer la sous-famille des images récursives qui n'utilisent pas la recursivité !)

2 Liens avec les CPOs

Nous noterons dans la suite $I = [0, 1[$, $\Gamma = I \times I$ et

$$N : \begin{cases} P(\Gamma)^4 \rightarrow P(\Gamma) \\ (x, y, z, t) \mapsto \frac{x+(0,1)}{2} \cup \frac{y+(1,1)}{2} \cup \frac{z+(1,0)}{2} \cup \frac{t}{2} \end{cases}$$

La définition des images que nous avons vue ci-dessus n'est que partiellement satisfaisante, en particulier subsistent de nombreuses ambiguités. Par exemple, une image telle que $x = N(x, x, x, x)$ n'a pour le moment aucun sens.

Une manière de lever ces ambiguités consiste à définir l'image comme la première composante du plus petit point fixe d'une fonction $F : P(\Gamma)^n \rightarrow P(\Gamma)^n$ ou n est le nombre de nœuds de l'image.

Une fois les nœuds de l'image numérotés de 1 à n (la racine portant le numéro 1), la fonction F est définie de la manière suivante :

$$F(x_1,\cdots,x_n) = (f_1(x_1,\cdots,x_n),\cdots,f_n(x_1,\cdots,x_n))$$

où f_i est associée au i-ème nœud de la manière suivante: si le nœud est une feuille, alors f_i est la fonction constante qui vaut soit $\emptyset$ soit Γ, et si le nœud possède des fils portant les numéros $k(1), k(2), k(3)$ et $k(4)$, alors $f_i(x_1,\cdots,x_n) = N(x_{k(1)}, x_{k(2)}, x_{k(3)}, x_{k(4)})$

Chacune des composante de F est évidemment croissante et continue, donc F l'est également, et F admet un unique plus petit point fixe, qui est égal à $P_1(lim_{n\to\infty}F^n(\emptyset))$ (cf. [1]). Ainsi, une image est définie sans aucune ambiguïté comme une partie de Γ.

3 Les images vues comme des automates

On peut représenter une image sous la forme d'un automate qui reconnait les points qui sont dans l'image. Pour cela, on va associer à chaque image un automate fini déterministe complet sur l'alphabet à quatre lettres $X = \{1,2,3,4\}$. On va associer à chaque point de Γ une partie de X^*. On définit pour cela les deux applications

$$\begin{array}{l} \delta : \Gamma \to X \\ \delta_{|[0,\frac{1}{2}[\times[\frac{1}{2},1[} \equiv 1 \\ \delta_{|[\frac{1}{2},1[\times[\frac{1}{2},1[} \equiv 2 \\ \delta_{|[\frac{1}{2},1[\times[0,\frac{1}{2}[} \equiv 3 \\ \delta_{|[0,\frac{1}{2}[\times[0,\frac{1}{2}[} \equiv 4 \end{array}$$

et

$$\begin{array}{l} \rho : \Gamma \to \Gamma \\ \rho_{|[0,\frac{1}{2}[\times[\frac{1}{2},1[} = (t \mapsto 2.t - (0,1)) \\ \rho_{|[\frac{1}{2},1[\times[\frac{1}{2},1[} = (t \mapsto 2.t - (1,1)) \\ \rho_{|[\frac{1}{2},1[\times[0,\frac{1}{2}[} = (t \mapsto 2.t - (1,0)) \\ \rho_{|[0,\frac{1}{2}[\times[0,\frac{1}{2}[} = (t \mapsto 2.t) \end{array}$$

et pour chaque point de Γ, on peut donc définir une suite $\gamma_n(x)$ de lettres de X avec $\gamma_n(x) = \delta(\rho^{n-1}(x))$. On a la relation $\gamma_n(\rho(x)) = \gamma_{n+1}(x)$. On note $L(x)$ le langage constitué des mots de la forme $\gamma(1)\cdots\gamma(n)$.

L'automate associé à l'image va avoir pour états les nœuds de l'image, et les deux couleurs noir et blanc, pour transitions les relations de descendances, et pour état terminal l'état noir. $\mathcal{A} =< X, Q, D, \{N\}, \delta >$ avec $Q = V \cup \{B, N\}$.

Par exemple, pour le triangle défini précédemment, l'automate $\mathcal{A}$ est représenté en figure 3.

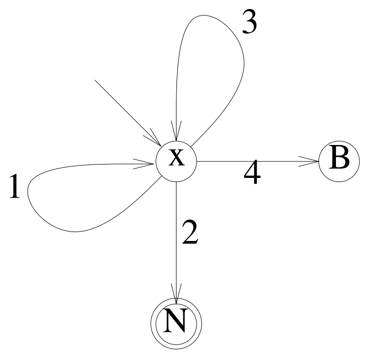


FIG. 3 - *Exemple d'automate associé à une image*

Réciproquement, si l'on se donne un automate fini déterministe complet sur un alphabet à quatre lettres qui possède un état poubelle et un état "puit" terminal, on peut lui associer une image récursive.

Si on parvient à démontrer une équivalence entre la définition par les CPOs et celle par les automates, on dispose d'une représentation concrète des images, représentation qui nous permet de réutiliser les résultats issus de la théorie des langages formels. En particulier, nous pourrons minimiser le nombre de nœds d'une image grâce à l'algorithme de Moore.

Proposition 3.1 *Soit $\mathcal{I}$ une image, $F : P(\Gamma)^n \rightarrow P(\Gamma)^n$ la fonction associée, et $\mathcal{A}$ son automate associé. Alors :*

$$\forall n \in N, (x \in P_1(F^n(\emptyset, \cdots, \emptyset))) \Longleftrightarrow (\gamma_1(x) \cdots \gamma_n(x) \in L(\mathcal{A}))$$

Preuve

On va procéder par récurence sur n.

$n = 1$

$$\begin{aligned}
(x \in P_1(F(\emptyset, \cdots, \emptyset))) &\Longleftrightarrow (x \in f_1(\emptyset, \cdots, \emptyset)) && (1)\\
&\Longleftrightarrow (f_1 \equiv \Gamma) \vee (x \in N(\emptyset, \emptyset, \emptyset, \emptyset)) && (2)\\
&\Longleftrightarrow (f_1 \equiv \Gamma) && (3)\\
&\Longleftrightarrow (\varepsilon \in L(\mathcal{A})) && (4)
\end{aligned}$$

$n > 1$

$$\begin{aligned}
(x \in P_1(F^n(\emptyset, \cdots, \emptyset))) &\Longleftrightarrow (f_1 \equiv \Gamma) \vee (\exists k \in N, \rho(x) \in P_k(F^{n-1}(\emptyset, \cdots, \emptyset))) && (5)\\
&\Longleftrightarrow (\varepsilon \in L(\mathcal{A})) \vee (\exists k \in N, x \in (\gamma_1(\rho(x)) \cdots \gamma_{n-1}(\rho(x)) \in L(\mathcal{A}'))) && (6)\\
&\Longleftrightarrow (L(\mathcal{A}) = X^*) \vee (\gamma_2(x) \cdots \gamma_n(x) \in L(\mathcal{A}')) && (7)\\
&\Longleftrightarrow (L(\mathcal{A}) = X^*) \vee (\gamma_1(x) \cdots \gamma_n(x) \in \gamma_1(x).L(\mathcal{A}')) && (8)\\
&\Longleftrightarrow (L(\mathcal{A}) = X^*) \vee (\gamma_1(x) \cdots \gamma_n(x) \in L(\mathcal{A})) && (9)\\
&\Longleftrightarrow (\gamma_1(x) \cdots \gamma_n(x) \in L(\mathcal{A})) && (10)
\end{aligned}$$

On déduit immédiatement de la proposition précédente le résultat suivant :

Proposition 3.2 *$x \in \mu(F)$ si et seulement si $\exists n \in N, \gamma_1(x) \cdots \gamma_n(x) \in L(\mathcal{A})$.*

On a donc pu implémenter la minimisation en appliquant simplement l'algorithme de Moore.

4 Types de données et algorithmes

4.1 types

Une image est donc équivalente à un automate, et les automates se représentent le plus souvent sous forme de graphe : un nœud par etat, et une flèche par transition.

4.1.1 type avec pointeurs

La représentation la plus traditionnelle des graphes fait appel à des pointeurs (*references* en Caml), elle s'applique donc tout naturellement ici. Une image est soit Blanc, soit Noir, soit une liste de pointeurs vers des images : ses fils. On y a ajouté une troisième possibilité, Fantome, pour les nœuds non-instanciés et dont on verra l'utilité plus loin.

```
type color = Blanc | Noir;;
type image = Fantome | Color of color | Node of (image ref) list;;
```

Le triangle cité en exemple est alors :

```
let rec tr=Node [ref (Color Blanc); ref tr; ref (Color Noir); ref tr];;
```

Ce type impose une bonne maîtrise de la différence entre égalité physique (= =) et égalité structurelle (=) avec les références, si l'on ne veut pas aboutir à des comparaison qui bouclent infiniment. Cela pose des problèmes dès que l'on veut comparer deux nœuds, en particulier, si l'on veut savoir si un nœud a déjà été rencontré dans un algorithme, il faut impérativement utiliser l'égalité physique sur l'objet, et non un test d'égalité sur les références comme on le ferait avec un langage classique.

4.1.2 type récursif

On peut aussi utiliser, sans grande différence conceptuelle, un type récursif avec des mutables :

```
type color   = Blanc | Noir;;
type image   = Color of color | Node of feuille
and  feuille = {mutable fils : image list};;
```

Et le triangle devient :

```
let rec triangle=Node {fils=[Color Blanc; triangle; Color Noir; triangle]};;
```

Il est absolument nécessaire d'utiliser des mutables pour pouvoir modifier une image, sinon la plupart des algorithmes ne seraient pas implémantables, faute de pouvoir construire une image progressivement. En effet, pour l'image du triangle par exemple, pour instancier la variable, il faut connaître ses fils, et comme elle fait partie de ses fils, il faut l'avoir déjà définie pour instancier ses fils ! (voir les algorithmes)

Ce type est particulièrement intéressant pour qui veut lui ajouter des caractéristiques par la suite, il est alors très simple d'ajouter des champs, ce qui a été fait pour la compression fractale.

4.1.3 type avec des tableaux

Il est aussi possible de numéroter chacune des variables de l'image et de les ranger dans un tableau. Une variable peut alors désigner ses fils par leurs indices dans le tableau si ce sont aussi des nœuds, et si ce sont des couleurs, on peut définir une convention comme -1 pour Noir, et -2 pour Blanc ; la racine de l'image est alors la première variable :

```
type variable== int list;;
type image==variable vect;;
```

(On aurait aussi pu choisir :

```
type fils = Var of int | Blanc | Noir;;
type variable== fils list;;
type image==variable vect;;
```

qui est plus propre, mais sûrement moins compact en mémoire)

Le triangle est alors :

```
let triangle=[|-2;0;-1;0|]
```

Ce qui est déjà moins lisible ! En revanche cette représentation non récursive a l'avantage de permettre un parcours facile et instinctif de l'arbre de l'image (voir plus loin les algorithmes) sans risquer de "boucler", il suffit en effet de parcourir le tableau. De même, tous les algorithmes qui ont besoin de stocker des références à une variable pourront le faire très facilement grâce aux indices, en particulier le calcul de la surface et la résolution de matrice qu'il entraîne. Malheureusement, c'est inefficace du point de vue de la gestion mémoire, les tableaux de longueur variable sont difficiles à gérer, et dans le cas de l'intersection on ne sait rien à l'avance du nombre de nœuds créés, si ce n'est qu'il est borné par n^2 !

L'efficacité du garbage-collector (euh, glaneur de cellules, pardon M. Toubon !) de Caml nous a donc amené à choisir le type avec références, qui est également plus parlant.

4.2 les algorithmes de base

4.2.1 zoom

Zoomer sur un des coins de l'image consiste simplement à renvoyer l'élément correspondant dans la liste des fils de l'image, en distinguant le cas d'une image unie bien entendu. La représentation en tableau imposerait alors, si l'on voulait avoir une gestion efficace de la mémoire, de regarder quels sont les éléments du tableau qui ne sont plus utilisés et de les en enlever, alors qu'ici le GC se charge de tout.

```
let zoom z = function
  Fantome -> failwith("Zoom sur une image fantome")
| Color c -> Color c
| Node l -> !(k\_ieme z l);;
```

Ce zoom est donc réalisé en temps constant.

4.2.2 inversion

Il suffit de vérifier que

$$\left\{\begin{array}{ccc} inverse\begin{pmatrix} A & B \\ D & C \end{pmatrix} & = & \begin{pmatrix} inverse(A) & inverse(B) \\ inverse(D) & inverse(C) \end{pmatrix} \\ inverse(Noir) & = & Blanc \\ inverse(Blanc) & = & Noir \end{array}\right.$$

Et on est ramené à échanger dans toutes les variables le Noir et le Blanc.

Par contre, le type tableau est ici plus simple : il suffit de parcourir le tableau et d'intervertir les -1 et les -2, alors qu'on est obligé, avec les références, à conserver une liste des variables déjà rencontrées pour éviter de boucler en parcourant l'arbre qui représente l'image. Cette technique est en fait à la base de la plupart des routines écrites, avec de petites variantes. C'est ici qu'apparaît l'intérêt du type Fantome, qui permet d'instancier une image dont on ne connaît pas encore les fils mais dont on veut avoir une référence (c'est également ici qu'apparait la nécessité des *mutables*.) On définit donc une référence sur un Fantome, et ensuite seulement on affecte à cette référence un Node avec la liste de ses fils (voir le code).

```
let inverse im=
  let deja\_vues = ref [] in
  let rec inverse\_aux = function
    (Color Blanc) -> ref (Color Noir)
  | (Color Noir)->ref (Color Blanc)
  | n{\oe}ud -> try assq n{\oe}ud !deja\_vues
        with Not\_found -> let h = ref Fantome
                   in deja\_vues := (neud, h)::(!deja_vues);
                         h := Node(map inverse\_aux (fils i));
                       h
in !(inverse_aux im);;
```

Comme l'algorithme consiste à parcourir le graphe, il est en $O(n)$.

Finalement la routine d'inversion a été généralisée, puisqu'elle prend comme argument une fonction color $\rightarrow$ image pour modifier chaque occurrence de Blanc ou Noir par l'image qui lui est associée.

4.2.3 pivotement

Le pivotement à 90° a été programmé ainsi lui aussi, en modifiant l'ordre des listes des fils de chaque variable, puisque

$$\left\{\begin{array}{ccc} R\begin{pmatrix} A & B \\ D & C \end{pmatrix} & = & \begin{pmatrix} R(B) & R(C) \\ R(A) & R(D) \end{pmatrix} \\ R(Noir) & = & Noir \\ R(Blanc) & = & Blanc \end{array}\right.$$

En terme de langage formel, cette routine revient a permuter les lettres de l'alphabet. On a en effet renommé toutes les transitions. Il s'effectue lui aussi en $O(n)$.

4.2.4 intersection

L'intersection relève elle aussi du même principe, en un peu plus compliqué, puisqu'ici deux images sont à considérer ; notons les A et B. Pour chaque couple de variable X et Y de A et B rencontré on crée une variable $X \cap Y$. On a en effet :

$$\left\{\begin{array}{ccc} \begin{pmatrix} X_1 & X_2 \\ X_4 & X_3 \end{pmatrix} \cap \begin{pmatrix} Y_1 & Y_2 \\ Y_4 & Y_3 \end{pmatrix} & = & \begin{pmatrix} X_1 \cap Y_1 & X_2 \cap Y_2 \\ X_4 \cap Y_4 & X_3 \cap Y_3 \end{pmatrix} \\ Noir \cap X & = & X \\ Blanc \cap X & = & Blanc \end{array}\right.$$

Les deux dernières relations permettent de simplifier le calcul par rapport au cas général d'une opération binaire sur deux image où il faudrait impérativement rencontrer deux couleurs pour donner le résultat, en s'appuyant sur la table de vérité de la relation. C'est pourquoi la procédure n'a pas été généralisée, contrairement a ce qui a par la suite été fait pour les images en couleurs. Et le programme termine forcément puisque le nombre de variable créées est majoré par nm+n+m où n est le nombre de variables de A et m celui de B (mn pour les $X \cap Y$, n pour $X \cap Noir$ et m pour les $Noir \cap Y$).

```
let intersection i j =
let deja_vues = ref [] in
let rec inter_aux = function
  Fantome -> failwith("Calcul d'intersection sur une image fantome")
| Color Blanc -> (function _ -> ref (Color Blanc))
| Color Noir  -> (function i -> ref i)
| i -> function
    Fantome -> failwith("Calcul d'intersection sur une image fantome")
  | Color Blanc -> ref (Color Blanc)
  | Color Noir  -> ref i
  | j -> try cpl_assq (i, j) !deja_vues
         with Not_found -> let h = ref Fantome
                           in deja_vues := ((i, j), h)::(!deja_vues);
                           h := Node(map2 inter_aux (fils i) (fils j));
                           h
in !(inter_aux i j);;
```

Proposition 4.1 *L'intersection de deux images A et B d'automates $\mathcal{A}$ et $\mathcal{B}$ par l'algorithme proposé calcule bien l'image C d'automate $\mathcal{C}$ tel que $L(\mathcal{C}) = L(\mathcal{A}) \cap L(\mathcal{B})$.*

Preuve

Soit $\mathcal{A} = \langle X, Q_A, D_A, \{N\}, \delta_A \rangle$

et $\mathcal{B} = \langle X, Q_B, D_B, \{N\}, \delta_B \rangle$

$\mathcal{C}$ a pour états $Q_A \times Q_B$. On définit la fonction de transition δ_C par :

$$\forall (q_A, q_B) \in Q_A \times Q_B,\ \delta_C(q_A, q_B) = (q'_A, q'_B) \iff \delta_A(q_A) = q'_A \text{ et } \delta_B(q_B) = q'_B.$$

On définit bien sûr l'état initial comme étant (D_A, D_B) et l'état final est (N, N).

Remarque *c'est bien ce que fait l'algorithme, avec un automate déterministe complet sur l'alphabet à quatre lettres considéré.*

Maintenant soit $w \in X^*$.

$$\begin{array}{rcl}
w \in L(\mathcal{C}) & \iff & w : D_C \xrightarrow{\delta_C} (N, N) \\
& \iff & \exists (D_A, D_B) = (p_1, q_1), \cdots, (p_k, q_k) = (N, N) \in Q_A \times Q_B \\
& & tq\ \forall i < k,\ \delta_C(p_i, q_i) = (p_{i+1}, q_{i+1}) \\
& \iff & \exists (D_A, D_B) = (p_1, q_1), \cdots, (p_k, q_k) = (N, N) \in Q_A \times Q_B \\
& & tq\ \forall i < k,\ \delta_A(p_i) = p_{i+1} \text{ et } \delta_B(q_i) = q_{i+1} \\
& \iff & \exists D_A = p_1, \cdots, p_k = N \in Q_A\ tq\ \forall i < k,\ \delta_A(p_i) = p_{i+1} \text{ et} \\
& & \exists D_B = q_1, \cdots, q_k = N \in Q_B\ tq\ \forall i < k,\ \delta_B(q_i) = q_{i+1} \\
& \iff & w : D_A \xrightarrow{\delta_A} N \text{ et } w : D_B \xrightarrow{\delta_B} N \\
& \iff & w \in L(\mathcal{A}) \text{ et } w \in L(\mathcal{B})
\end{array}$$

Et on l'a vu il s'agit d'un algorithme en $O(nm)$.

4.3 Le calcul de la surface

Pour calculer la surface d'une image, on pose

$$\left\{ \begin{array}{rcl}
S(Blanc) & = & 0 \\
S(Noir) & = & 1 \\
S\begin{pmatrix} A & B \\ C & D \end{pmatrix} & = & \frac{1}{4}[S(A) + S(B) + S(C) + S(D)]
\end{array} \right.$$

Chaque variable de l'image voit donc sa surface s'exprimer par une équation linéaire sur la surface de ses fils. Pour calculer la surface de l'image, on doit résoudre le système linéaire ainsi formé. Chaque ligne représente l'équation associée à une variable, il est donc nécessaire d'indexer celles-ci, ce qui aurait été simplifié dans le cas de la représentation en tableaux. On commence donc par associer à chaque variable un entier en parcourant simplement l'arbre, puis on remplit la matrice en regardant l'indice des fils de chaque variable, les termes constants proviennent des éventuels fils Noir. Le système est ensuite résolu par la méthode du pivot de Gauss sur les matrices réelles. Cependant, on peut remarquer que la matrice ne comporte au plus que cinq coefficients non nul (le terme diagonal et les quatre fils) par ligne en comptant le terme constant. La procédure vérifie donc si le coefficient est non nul et différent de 1 avant d'effectuer un calcul coûteux. On obtient ainsi la surface de toutes les variables, ce qui s'avère très utile pour l'affichage comme on le verra par la suite.

Une autre méthode avait tout d'abord été implémentée qui tirait parti de la structure en arbre et consistait à simplifier le système à chaque nœud. En effet on implante une fonction récursive *equation_surface* qui prend pour argument la variable X dont on veut la surface et un ensemble $\{Y_i\}$ de variables qui sont celles situées au dessus de X dans l'arbre, et qui renvoie la surface de X comme combinaison linéaire des Y_i.

```
surface X {Yi} =
  si X=Blanc renvoie 0
  si X=Noir renvoie 1
  si il existe i tq X=Yi alors renvoie Yi
  sinon
    combinaison_lineaire=0
    pour chaque    fils Z de X
      ajoute 1/4(surface Z {Yi}UX} a combinaison_lineaire
    match combinaison_lin\'eaire with aX+b+Somme(aiYi)
          et renvoie b+Somme(aiYi)/(1-a)
```

(en effet on a

$$S(X) = \frac{1}{4}(\sum_{Z fils de X} S(Z) = combinaison\_linaire$$

d'où le résultat renvoyé qui est la surface de X en fonction des seules variables qui sont au dessus de lui.)

Malheureusement, cet algorithme ne fonctionne plus dès que l'on autorise les variables utilisées à ne pas se trouver uniquement au dessus dans l'arbre, il a donc été abandonné ; il avait également le défaut de ne calculer, dans un premier temps, que la surface de l'image toute entière (on pouvait tout de même calculer simplement les surfaces des variables en parcourant l'arbre à nouveau, ce qui revenait à résoudre le système en cascade).

Toutefois, étant donné la spécificité de la matrice (une majorité de 0, et uniquement des coefficients $-\frac{1}{4}, -\frac{1}{2}, -\frac{3}{4}, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}$ et 1 !), la matrice peut très certainement être inversée de manière plus élégante et plus juste, malheureusement, cette possibilité n'a pu être exploitée faute de temps.

4.4 Affichage

4.4.1 new_printer

Pour les structures récursives infinies, Caml se contente d'afficher les premiers niveaux, puis termine par des points de suspension évasifs, ce qui donne pour notre triangle fétiche :

```
triangle : image = Node [ref (Color Blanc); ref (Node [ref (Color Blanc); ref (Node [ref (Color Blanc);
ref (Node [ref (Color Blanc); ref (Node [ref (Color Blanc); ref (Node [ref (Color Blanc); ref (Node [ref
(Color Blanc); ref (Node [ref (Color Blanc); ref (Node [ref (Color Blanc); ref (Node [ref (Color Blanc);
ref (Node [ref (Color Blanc); ref (Node [ref (Color Blanc); ref (Node [ref (Color Blanc); ref (Node [ref
(Color Blanc); ref (Node [ref (Color Blanc); ref (Node [ref (Color Blanc); ref (Node [ref (Color ...
```

On a donc utilisé la fonction *new_printer* pour redéfinir cet affichage. On commence tout d'abord par numéroter toutes les variables comme pour la surface, puis les variables sont affichées sous la forme :

n° : (n° du 1er fils ; n° du 2^{e} fils ; ... ; n° du n^{e} fils)

Bien sûr, si le fils est Blanc ou noir, ce n'est pas son numéro qui est affiché mais B ou N. Ce qui donne :

```
triangle : image = 0 : (B; 0; N; 0);
```

4.4.2 affichage graphique

Une procédure d'affichage graphique a par ailleurs été développée. Programmée de manière récursive, elle se contente de parcourir l'arbre en profondeur en s'arrêtant à un niveau déterminé, en général la taille d'un pixel (point affiché à l'écran), et dans un premier temps on choisit de ne rien afficher dans ce cas. Seuls sont donc visibles les carrés noirs plus grands qu'un pixel.

```
let rec affichage mini x y l = function
```

```
  (Color Noir)-> fill\_rect x y l l
| (Color Blanc) -> ()
| i -> if l>mini
      then match (fils i) with
           [c1; c2; c3; c4]->
                  let l2=l/2 in
                  affichage  mini x (y+l2) l2 c1;
                  affichage mini (x+l2) (y+l2) l2 c2;
                  affichage mini (x+l2) y l2 c3;
                  affichage mini x y l2 c4;;
```

La procédure effectivement utilisée est un peu plus compliquée, puisqu'elle gère la possibilité d'une taille d'image qui ne soit pas une puissance de 2, ce qui peut entraîner que tous les fils d'une variable n'aient pas exactement la même taille ($\frac{l}{2}$ et $l - \frac{l}{2}$), de ce fait les dimensions verticales et horizontales ne sont plus forcément les mêmes, mais tout cela ne complique pas beaucoup le programme.

Cependant l'affichage dans des fenêtres dont la taille n'est pas une puissance de 2 présente une bien moindre qualité, en raison de la nature même des images.

4.4.3 antialiasing

Cependant, l'affichage en blanc des variables de la taille d'un pixel comporte de nombreuses limitations : tout d'abord les traditionnels effets d'escalier, ou aliasing (crênelage, même si ce n'est pas du tout la traduction exacte), et surtout, si les images ne sont pas bien définies, on a des effets peu souhaitables (par exemple de l'union des deux triangles, où une image qui devrait être entièrement noire a une diagonale blanche.) C'est pourquoi on a fait appel aux technique d'anti-aliasing (ou anticrênelage, ou même, paraît-il, linéarisation !). Le principe est simple : au lieu de se limiter au noir et au blanc, on pallie la faible résolution des écran par une utilisation judicieuse des niveaux de gris qui permet de lisser les images affichées. La surface des variables a été utilisée pour déterminer la nuance de gris utilisée. La seule modification à apporter à la procédure d'affichage est donc, dans le cas où on atteint la taille du pixel, d'afficher un point de ce niveau de gris.

Cet anticrênelage est toutefois le plus simple et le moins efficace qui existe, en effet, dans le cas optimal, toutes les parties du point n'ont pas la même influence (schématiquement proportionnellement à leur distance au centre du pixel), et, par ailleurs, les zones d'influence devraient se chevaucher. Il eût été possible de satisfaire un peu mieux la première exigence, en subdivisant encore un peu après avoir atteint la taille du pixel, quant à la seconde, elle s'adapte beaucoup moins à la structure récursive de l'affichage.

4.4.4 affichage de la structure

Une autre routine a été programmée qui permet de se rendre compte de la structure de l'image : elle entoure en effet chaque variable d'un cadre rouge.

4.4.5 impression Postscript

La procédure d'affichage a également été adaptée pour permettre l'impression des images sur imprimante PostSript, en utilisant la puissance de ce langage. On génère en effet pour toutes les variables de l'image une procédure Postscript qui appelle récursivement les procédures des autres variables. Pour cela les variables sont numérotées selon la technique habituelle, et Caml écrit la procédure en regardant le numéro des fils (ou en ne faisant rien si c'est Blanc et en affichant un carré si c'est Noir). Les techniques d'anticrênelage n'ont pas été employées, puisque les imprimantes sont généralement noir et blanc et utilisent le procédé inverse pour simuler les niveaux de gris, à savoir une diminution de la résolution et des trames.

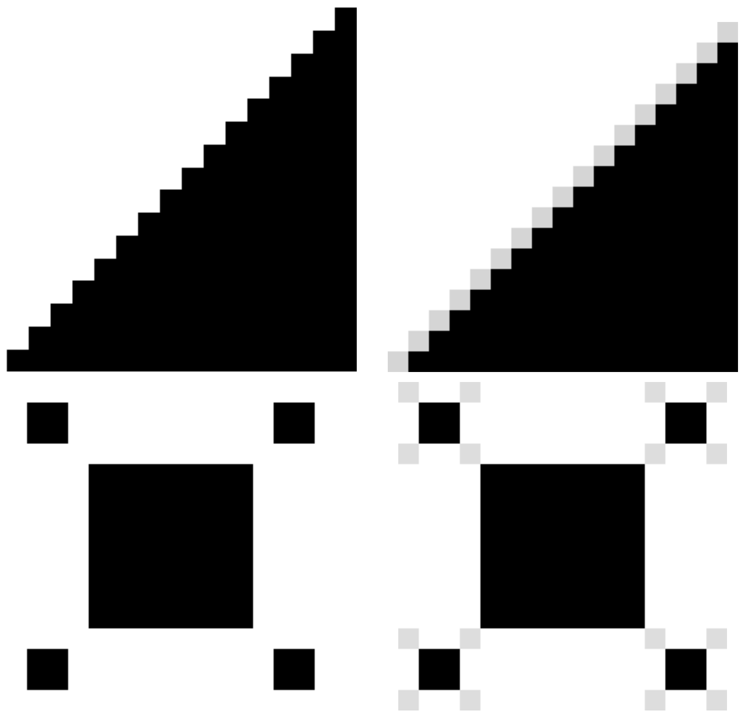


FIG. 4 - *Affichage sans et avec antialiasing*

5 Géometrie rationnelle

5.1 demi-plan oblique

L'introduction de la recursivité permet de représenter parfaitement des parties de Γ obtenue par intersection de Γ avec un demi-plan défini par une équation affine à coefficients rationnels. En effet, soit $I = \Gamma \cap P$ où $P : a.x + b.y + c \geq 0$ est un tel demi-plan (a, b, c entiers relatifs), alors il y a trois cas possibles. Les deux premiers cas sont $\Gamma \subset P$, alors $I = \Gamma$ et $\Gamma \subset P^c$, alors $I = \emptyset$.

Dans le troisième cas, Γ est en partie contenu dans P, or, si on pose $\Gamma = N(\Gamma, \Gamma, \Gamma, \Gamma)$, alors $\Gamma \cap P = N(\Gamma \cap P_1, \Gamma \cap P_2, \Gamma \cap P_3, \Gamma \cap P_4)$, où les P_i sont définis par des équations de la forme $a.x + b.y + c_i \geq 0$. Or, si les quatre sommets de Γ sont contenus dans P (respectivement en dehors de P), alors Γ est tout entier contenu dans P (respectivement en dehors). Donc, si $a + b + c_i \geq 0$, $a + c_i \geq 0$, $b + c_i \geq 0$, et $c_i \geq 0$, alors $\Gamma \cap P_i = \Gamma$ (respectivement, si $a + b + c_i \leq 0$, $a + c_i \leq 0$, $b + c_i \leq 0$, et $c_i \leq 0$, alors $\Gamma \cap P_i = \emptyset$, modulo un ensemble de mesure nul).

Donc $\Gamma \cap P_i \notin \{\emptyset, \Gamma\}$ si et seulement si $Min(a + b, a, b, 0) < -c_i < Max(a + b, a, b, 0)$. Or $Min(a+b, a, b, 0) = Min(a, 0) + Min(b, 0) = -(a^- + b^-)$ et de même $Max(a+b, a, b, 0) = a^+ + b^+$, donc $Max(a + b, a, b, 0) - Min(a + b, a, b, 0) = |a| + |b| - 1$

Finalement, si on montre que les c_i sont entiers, on aura montré que l'arbre qui représente I a au plus $|a| + |b| - 1$ nœuds, ou encore qu'on peut effectivement représenter I avec un arbre quaternaire.

Or, si x, y sont les coordonnées dans Γ et x', y' celles dans l'un des quatre "sous-Γ", on a $x = \frac{x'}{2} + \alpha, y = \frac{y'}{2} + \beta$ où $\alpha, \beta = 0$ ou 1.

Donc l'équation de P devient

$a.(\frac{x'}{2} + \alpha) + b.(\frac{y'}{2} + \beta) + c \geq 0$ c'est à dire

$a.(x' + 2.\alpha) + b.(y' + 2.\beta) + 2.c = a.x' + b.y' + 2.(a.\alpha + b.\beta + c) \geq 0$

Donc $c' = 2.(a.\alpha + b.\beta + c)$, qui est bien un entier relatif.

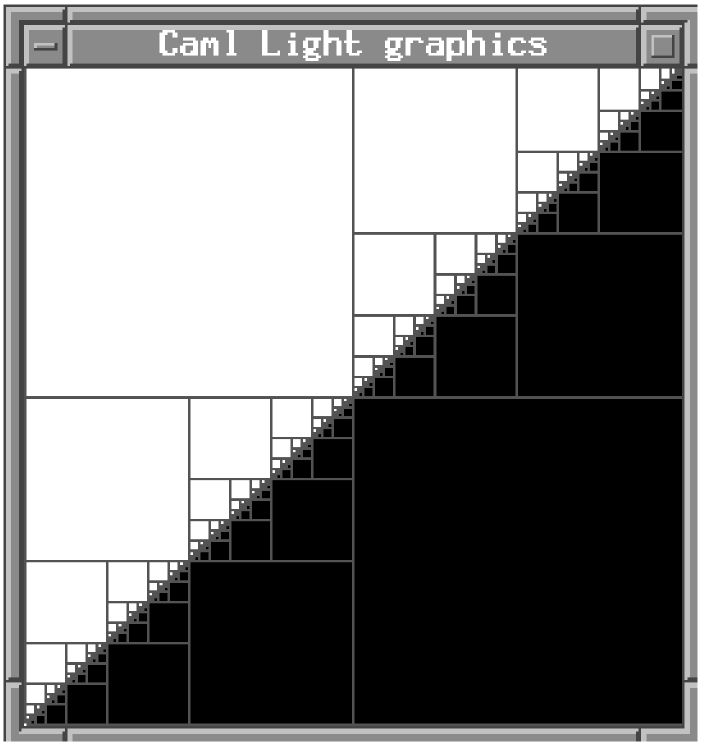


Fig. 5 - *affichage de la structure d'une image*

La routine CAML qui construit un demi-plan oblique à partir des coefficients a, b et c est la suivante :

```
(* partie de la forme a*x+b*y+c>=0 avec a, b et c entiers relatifs *)

let oblique a b c =

let deja_vues = ref []
and mn = min (min a b) (min 0 (a+b))
and mx = max (max a b) (max 0 (a+b))
in

let rec oblique_aux = function
```

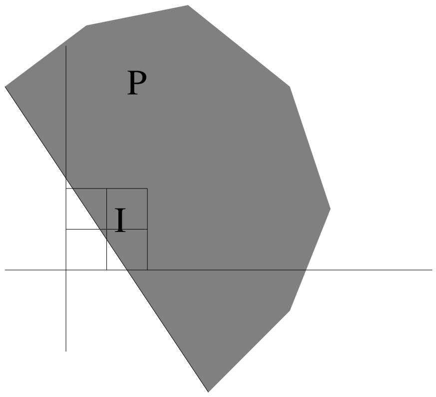


Fig. 6 - *Intersection de Γ et d'un demi-plan*

```
  c -> if c+mn>=0 then ref (Color Noir)
       else if c+mx<=0 then ref (Color Blanc)
            else try assq c !deja_vues with
                 Not_found -> let h = ref Fantome
                              in deja_vues := (c, h)::(!deja_vues);
                                 h := Node [ oblique_aux (2*c);
                                             oblique_aux (a+2*c);
                                             oblique_aux (a+b+2*c);
                                             oblique_aux (b+2*c) ];
                                 h

in !(oblique_aux c)
;;
```

Ainsi, sachant représenter l'union et l'intersection de deux images, ainsi que les demi-plans rationnels, on est en mesure de calculer l'arbre associé à n'importe quel polygone dont les cotés sont définis par des coordonnées rationnelles..

FIG. 7 - *exemple de polygone*

5.2 transformation affine

le principe de la transformation affine est fort simple : on spécifie un parallèlogramme dans le plan de l'image de départ, et qui correspondra à l'image d'arrivée. Le parallèlogramme est définit à l'intérieur d'un bloc carré de quatre images (A,B,C,D) par ses coordonnées rationnelles : $(\frac{x}{2d}, \frac{y}{2d})(\frac{u_x}{2d}, \frac{u_y}{2d})(\frac{v_x}{2d}, \frac{v_y}{2d})$ (voir figure **??**) On impose par ailleurs
$\frac{d}{2} < max\{$largeur du parallèlogramme, hauteur du parallèlogramme $\} < d$.

On a alors deux cas : soit les quatre images sont toutes noires ou toutes blanches et l'image renvoyée est Noir ou Blanc, soit on subdivise le parallélogramme en quatre, et on rappelle la procédure avec chacun des quatre fils avec les quatre variables (parmi les fils des variables précédentes) qui le contiennent, et les valeurs x' et y' mises à jour.

x et y restent entiers car le choix des quatre fils qui contiennent le sous parallèlogramme ne permet que trois possibilités pour x' :

$$\begin{array}{rcll} \frac{x'}{2d} & = & 2\frac{x}{2d} & \text{si le sous parallèlogramme est contenu dans la bande } 0 \leq x \leq \frac{1}{2} \\ \frac{x'}{2d} & = & 2(\frac{x}{2d} - \frac{1}{4}) & \text{si le sous parallèlogramme est contenu dans la bande } \frac{1}{4} \leq x \leq \frac{1}{4} \\ \frac{x'}{2d} & = & 2(\frac{x}{2d} - \frac{1}{2}) & \text{si le sous parallèlogramme est contenu dans la bande } \frac{1}{2} \leq x \leq 1 \end{array}$$

Et x' reste entier puisqu'on a 2d au dénominateur, donc ppcm(2,2d)=2d.

On a bien sûr les mêmes résultats pour y'.

Le nombre de variables créées est donc majoré par $n^4(2d)^2$ (ce qui correspond au nombre de quadruplets de quatre variables associés à deux entiers positifs inférieurs à 2d.) Le programme termine donc, mais est virtuellement en $O(n^4)$.

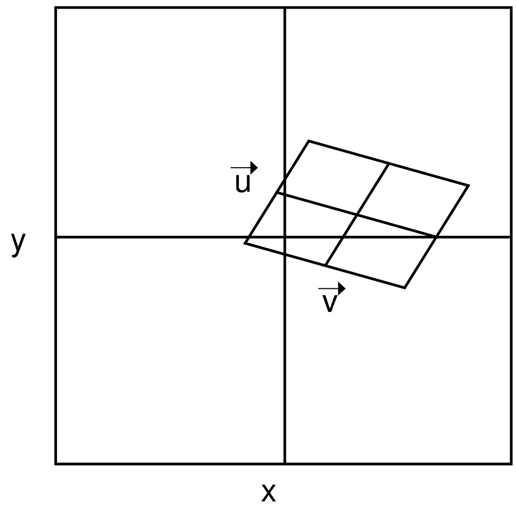


FIG. 8 - *Principe de la transformation affine*

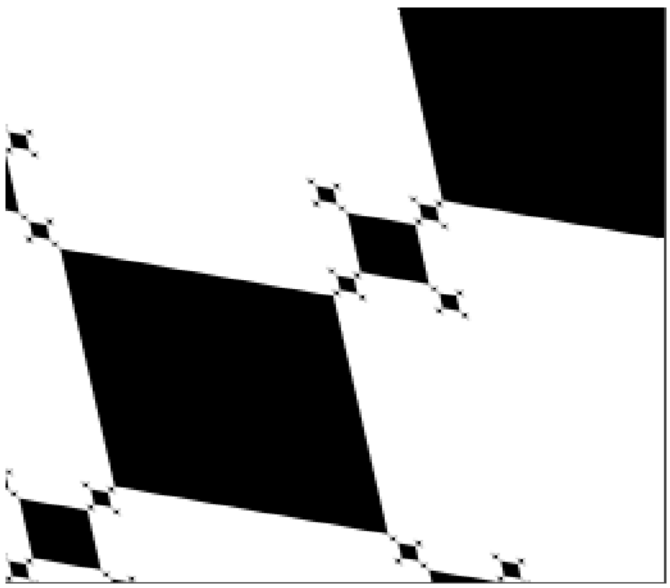


FIG. 9 - *affine 0 0 5 1 1 5 16 Sierp*

6 Perspectives

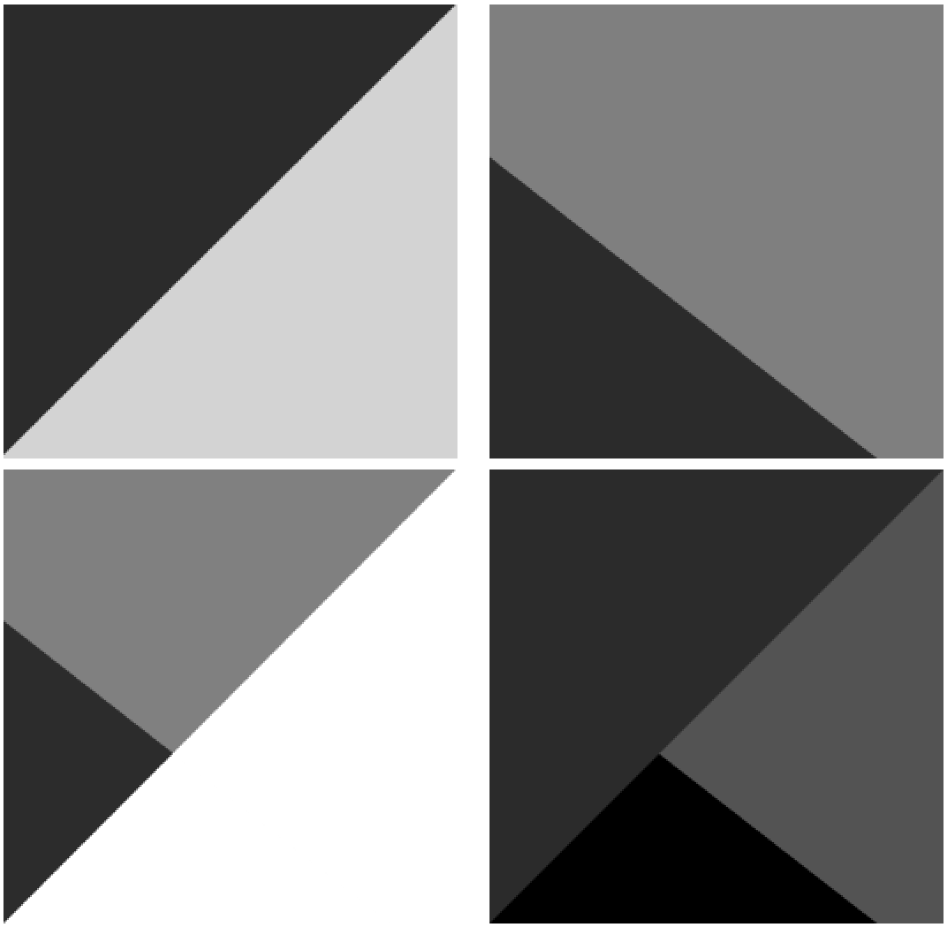
6.1 Couleur

Pour traiter une plus grande diversité d'images, on peut souhaiter étendre le modèle en y introduisant la couleur, ce qui ne modifie que très peu les routines employées.

```
type image = Fantome | Color of color | Node of (image ref) list;;
```

Où color est le type du module graphique de Caml.

Tout d'abord, une signification doit être donnée au calcul de l'intersection de deux images. Ici on a choisi de fournir à la routine une fonction color*color$\rightarrow$ color, qui permet de l'utiliser sur un grand nombre de transformation. On a par exemple implanté une superposition soustractive et une additive (voir figure **??**). Par ailleurs, on n'a plus la possibilité, comme en noir et blanc, de simplifier le calcul grâce aux relations : $Blanc \cap X = Blanc$ et $Noir \cap X = X$, il faut poursuivre jusqu'à rencontrer deux couleurs.



intersection additive — intersection soustractive

FIG. 10 - *Différentes intersections en couleur*

On est également amené à se demander ce que signifie calculer la surface d'une image en couleur. Une première interprétation pourrait être la surface non blanche (ou non noire), mais la plus intéressante, pour l'affichage en particulier, est sans doute la couleur moyenne de l'image, ce qui est également le cas en noir et blanc comme on l'a vu pour l'affichage anticrênelé. La matrice à inverser se construit de la même façon que précédemment, sauf en ce qui concerne les termes constants qui sont triplés et correspondent au niveau de chaque composante RVB (rouge, vert bleu), le système est ensuite résolu trois fois, une par composante, mais les calculs sont bien entendu effectués en parallèle. Cela nous permet de programmer l'affichage avec anticrênelage, exactement comme en noir et blanc, avec les mêmes limitations bien entendu.

La minimisation n'a pu être adaptée, faute de temps, c'est néanmoins tout à fait possible, le principal problème survient pour la phase préliminaire qui vérifie qu'une image est bien fondée et y remédie le cas échéant. En effet, s'il est très simple, en regardant sa surface, de voir si une image noir et blanc est uniforme, ça l'est beaucoup moins pour une image couleur : comment savoir si une image en moyenne grise est uniformément grise ou moitié noire et moitié blanche ? Il faut alors parcourir l'arbre, et regarder si plusieurs couleurs différentes sont rencontrées. De toute façon cette méthode est également la meilleure dans le cas monochrome, puisqu'elle ne souffre pas des erreurs de calcul en flottants, contrairement á l'inversion de la matrice par la méthode du pivot de Gauss.

6.2 octrees dans l'espace tridimensionnel et ray-tracing

On n'est pas non plus obligé de se limiter au plan, les octrees (arbres octaires ?) sont d'ailleurs très populaires en image de synthèse tridimensionnelle, toujours pour leur capacité à modéliser un objet à plusieurs précisions.

Une petite routine d'affichage en 3D cavalière a été programmée pour étudier ces possibilités, elle reprend exactement l'algorithme bidimensionnel, en considérant huit fils et en introduisant un décalage en diagonal pour simuler la profondeur, et au lieu d'afficher des carré, elle affiche des cubes.

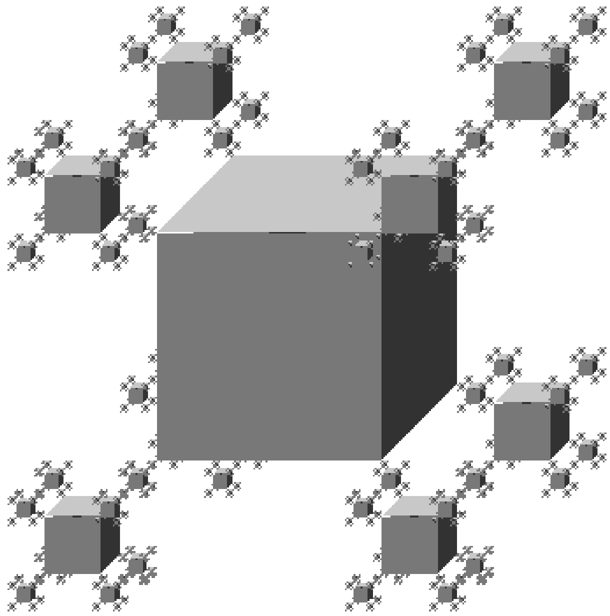
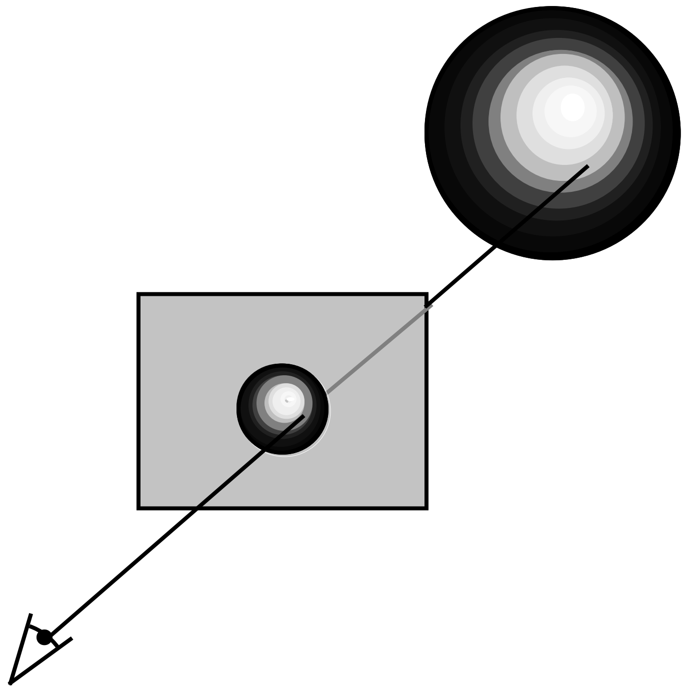


FIG. 11 - *Exemple d'octree récursif*

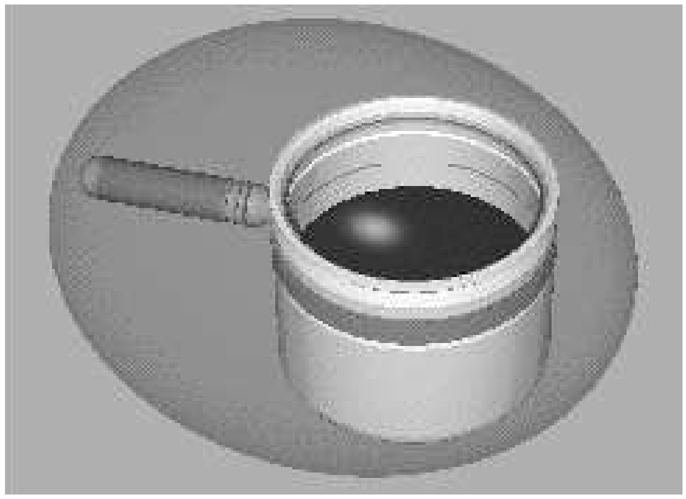
Les algorithmes de zoom, d'inversion, d'intersection n'ont pas à être modifiés, et pour que la surface fonctionne (et calcule le volume) il suffit de changer la valeur de *nombre_de_fils*.

L'un des algorithmes qui a le plus popularisé les octrees est très certainement le ray-tracing ou suivi de rayon, qui consiste à considérer le trajet inverse de la lumière en partant de l'œil de l'observateur et en regardant quel objet est rencontré en premier.

Un algorithme de ray-tracing passe donc la majeure partie de son temps à effectuer des calculs d'intersection entre les rayons (des droites) et les objets de la scène. Afin de simplifier ces calculs, on fait souvent appel à des volumes englobants, qui contiennent les objets considérés, mais dont



Principe du suivi de rayon



Principe des volumes englobants

l'intersection est plus rapide à calculer. Ainsi, si le volume englobant n'est pas intersecté, l'objet contenu non plus.

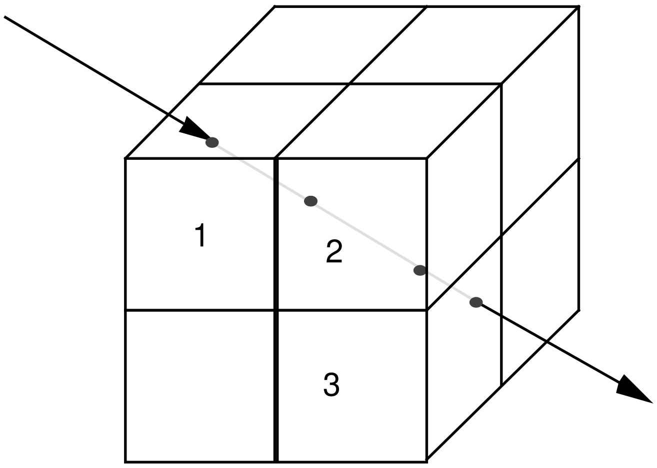
En général on utilise des octrees comme volumes englobants, leur structure hiérarchique permettant d'affiner peu à peu la région de l'espace considérée ; cependant ici on propose d'utiliser les octrees récursifs pour la modélisation d'objets tridimensionnels.

L'algorithme d'intersection avec un rayon consisterait simplement à considérer chaque cube que rencontre le rayon, s'il est blanc il n'y a pas d'intersection, s'il est noir il y a intersection et si c'est un nœud on appelle récursivement la procédure pour chacun des huit fils que rencontre le rayon, dans l'ordre. Là aussi, il est nécessaire de se fixer une limite pour la récursivité, mais cette limite peut varier en fonction de la distance de l'objet : on peut ainsi s'arrêter lorsque le cube considéré est plus petit que le pixel calculé, et profiter d'une modélisation détaillée au premier plan et de calculs plus rapides pour les objets lointains.

Le problème qui se pose alors est celui du calcul du vecteur normal à l'objet ; celui-ci est en effet à la base de tous les modèles d'éclairement (modèles qui permettent de calculer la couleur que perçoit l'œil en un point en fonction de l'éclairage et de la surface elle-même). Or les objets fractals ont ceci de désagréable que le vecteur normal n'y est souvent pas défini (par exemple sur l'image *Sierp3D*). On en revient au problème des bords de nos images.

On peut s'inspirer des techniques utilisées pour représenter en particulier l'ensemble de Mandelbroot en relief, et qui consistent à utiliser les points avoisinants dans l'image calculée. En effet on connaît très facilement les coordonnées (x,y,z) d'un point de l'image affichée, et si l'on calcule un point (u, v) à l'écran, il suffit de calculer le vecteur normal au plan défini par les points dans l'image tridimensionnelle qui correspondent à (u, v), (u-1, v) et (u, v-1). On peut bien sûr affiner le procédé en effectuant une moyenne sur les quatre points voisins.

Malheureusement, contrairement au cas bidimensionnel, il est plus délicat d'étendre ce modèle à plus de deux couleurs et encore plus à plusieurs textures (une texture modélise les caractéristique d'une surface : elle en définit la couleur, la transparence, etc.), car il est alors très difficile de décider de l'attitude à adopter quand on arrête le récursion, les techniques d'anticrênelage utilisées dans le



Intersection d'un rayon avec un octree


Probème de definition du vecteur normal

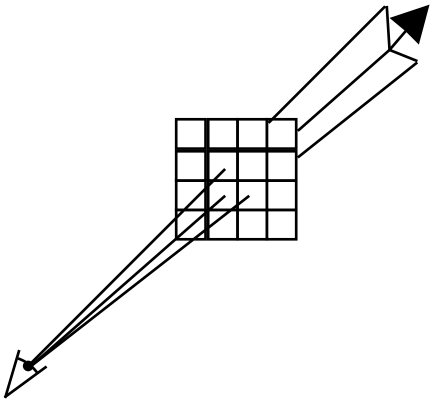


FIG. 12 - *Calcul du vecteur normal à l'aide des points avoisinants dans l'image écran*

plan se généralisent plus difficilement aux textures et à la 3D.

7 Compactage

7.1 bref historique

M. Barnsley a proposé une méthode de génération d'images basée sur les IFS (Iteretated Function System), et qui consiste a itérer plusieurs fois une transformation contractive à une image de départ (cf [2]). En vertu du théorème du point fixe, il y a convergence vers une image indépendante de l'image de départ. Il utilise pour ce faire un ensemble de transformations affines : $w(X) = \cup w_i(X)$.

Si s est le coefficient de contraction, d la métrique utilisée, le théorème de collage affirme même que

si

$$h(T, \bigcup_{n=0}^{N} w_n(T)) < \epsilon$$

alors

$$h(T, A_C) < \frac{\epsilon}{1-s}$$

où A_C désigne l'attracteur (point fixe) de $\{w_n : n = 0, 1, 2, \cdots, N\}$.

Pour approximer une image par ce moyen, il faut donc trouver l'ensemble de transformations affines par lequel l'image est "presque invariante". Cette méthode, manuelle, a permi d'obtenir des images saisissantes, en particulier la célèbre fougère de Barnsley qui est obtenue avec seulement 4 transformations.

Il proposa donc d'utiliser cette méthode pour compacter des images, mais elle nécessitait l'assistance d'un opérateur humain (d'où son nom d'*algorithme du graduate student*, elle consistait en effet à enfermer un étudiant avec une image et une station de travail puissante afin qu'il trouve les transformations), prenait beaucoup de temps de calcul, et il fallait également énormément de temps pour décompacter les images obtenues (de l'ordre de plusieurs heures sur des ordinateurs musclés !).

C'est anecdotiquement un des étudiants de Barnsley, Jaquin, qui proposa une solution sous la forme de PIFS (Partitionned Iterated Function System), c'est à dire que les transformations affines considérées sont restreintes à des portions du plan, et il implémenta un algorithme de compression fractale qui consistait, schématiquement, à partitionner l'image en blocs de 8*8 pixels, et de trouver, pour chaque bloc, un carré de 16*16 pixels qui lui soit similaire (en autorisant les isométries du carré et une transformation affine sur les niveaux de gris). Cette méthode a l'immense mérite d'être entièrement automatique et de présenter des temps de décompactage très raisonnables. En revanche, le temps de compression est très élevé, et la plupart des recherches actuelles visent à le réduire.

Nous nous devons de consacrer ici un paragraphe au "benchmark" officiel de la compression d'image, la célèbre Lenna (Lena en suédois). Il s'agit de la page centrale du numéro de novembre 1972 de Playboy. Lena Sderberg (née Sjooblom) est mariée et mère de trois garçons et a été fort surprise de sa célébrité informatique, et Playboy finirait quant à lui par se plaindre de cette violation répétée de ses copyrights.

7.2 implémentation

On a ajouté des opérateurs au quadtrees en niveaux de gris qui consistent en une isométrie du carré et une transormation affine sur les niveaux de gris. La métrique choisie est bien sûr l'ecart type, et pour que les transformations soient contractives, il suffit d'imposer que le facteur d'echelle de la transformation des niveaux de gris soit inférieur à 1.

```
deja_fait=[image_entiere]
a_faire=[image_entiere]
```

```
pour chacun des fils A de chaque variable de a_faire
  pour chaque variable B de deja_fait
    trouver la transformation f sur les niveaux de gris
    pour chaque isometrie g du carre
      calculer d(A-fog(B)) dans l'image bitmap
  si min d<seuil alors A=fog(B)
  sinon mettre A dans a_faire et deja_fait.
```

la transformation sur les niveaux de gris est calculée pour faire correspondre la moyenne et l'écart type des deux blocs. Les écarts types sont calculés pour la même taille d'image, c'est à dire que si l'on compare un bloc 8*8 et un bloc 16 *16, il faut ramener ce dernier à un bloc 8*8, pour ensuite seulement calculer l'écart type. L'algorithme a été amélioré en effectuant un test sur des réductions 4*4 des images, test qui échoue directement dans 50% des cas ce qui est loin d'être négligeable. On peut par ailleurs imposer une taille minimale pour les variables, ainsi, la plupart des images ont été calculées avec une taille minimale de 4, c'est à dire qu'au delà de cette valeur, même si la meilleure transformation est en dessus du seuil, on ne crée pas de nouvelle variable.

Le codage compact des images n'a pas été implémenté, c'est pourquoi seule une estimation du taux de compression est donnée.

FIG. 13 - *image originale, puis comprimée avec un taux supposé d'environ 10:1*

La nature recursive des images ainsi créées permet de les afficher à n'importe quelle résolution, il y a généralement création de détails à toutes les echelles, et en particulier les bords des images ne présentent pas les habituels effets d'escaliers des zooms sur les images bitmap. Cependant, ne rêvons pas, les détails créés ne correspondent pas exactement à la réalité, et il est hors de question d'espérer voir les atomes ou même les pores de la peau en zoomant suffisament. Cette astuce explique néanmoins les taux de compression exceptionnels affichés par certaines équipes qui, en effectuant un zoom de 4 fois sur leurs images, prétendent avoir obtenu un facteur de compression 16 fois meilleur que celui réellement calculé.

FIG. 14 - *structure de l'image compactée (seuls les nœuds non-transformés sont entourés)*

FIG. 15 - *zoom 2 fois*

7.3 optimisation et développements futurs

Comme il a été dit plus haut, la plupart des travaux actuels cherchent à accélérer la phase de compression, et l'une des méthodes employées est la classification des blocs. Par exemple Jaquin les classe selon qu'ils contiennent ou non des bords, d'autres [4] selon le nombre de niveaux de gris "dominants" qu'ils contiennent.

Il serait certainement possible d'utiliser également le rapport des écarts-types à différentes échelles : en effet considérons les deux blocs de la figure **??**. Le rapport des écarts types est très différent, les deux images n'ont donc pas à être comparées. Cette classification devrait permettre de distinguer les blocs comportant des hautes et des basses fréquences.

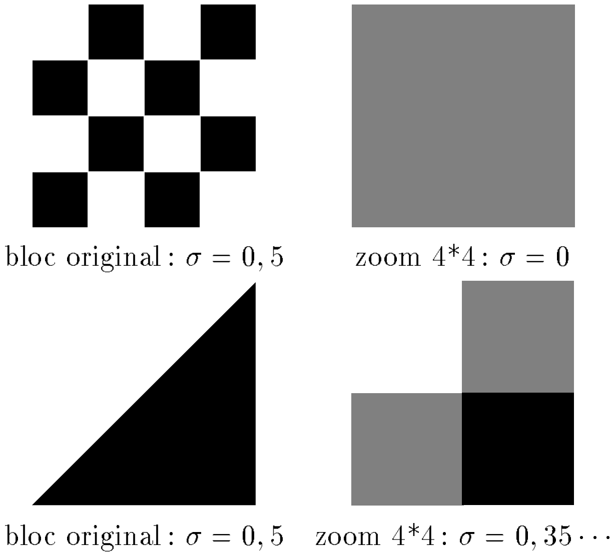


FIG. 16 - *Classification des images selon le rapport des écarts types*

8 Bibliographie

[1] B.A.Davey et H.A.Priestley *Introduction to lattices and order* Cambridge Mathematical Textbooks

[2] Michael F. Barnsley, Arnaud Jacquin, François Malassenet, Laurie Reuter et Alan D. Sloan *Harnessing Chaos for Image Synthesis* Computer Graphics, Volume 22, Number 4, August 1988.

[3] Yuval Fisher *Fractal Image Compression* Siggraph '92 Course Notes.

[4] Carsten Frigaard, Jess Gade, Thomas T. Hemmingsen et Torben Sand *Image Compression Based on a Fractal Theory* 1994.

[5] Michael F. Barnsley et Lyman P. Hurd *Fractal image compression* AK Peters.