Université de Genève
Faculté des Sciences
Département d'Informatique

SPIP

Search Picture In Pictures

Travail de diplôme
Marc Martin

Prof. Thierry Pun
Ass. Christian Rauber

Janvier/Juin 1995

Rapport théorique

Enoncé

Recherche d'images par leur contenu

Projet de diplôme ès-science informatique pour M. Martin.
Professeur responsable: Thierry Pun.
Assistant responsable: Christian Rauber.

Début du projet: 16 janvier 1995.

Résumé: Le but du projet est de rechercher dans une grande base de donnée d'images celles correspondant le mieux à un ou plusieurs critères visuels tels que, par exemple, la texture, la couleur ou les contours. La recherche se fait donc selon le contenu des images.
Le logiciel réalisé devrait être aisément utilisable pour la sélection de tissus dans l'ameublement ou la décoration d'intérieur.

Fonctionnalités attendues:

- interface graphique conviviale sous Windows;
- rajout aisé de critère de recherche dans le système;
- rapidité de la recherche;

Environnement de travail:

- programmation sur PC;
- programmation en C;
- compilateur: Borland C/C++ 3.1;
- programmation sous Windows 3.1;

Références:

- C. Faloutsos, M. Flickner, W. Niblack, D. Petkovic, W. Equitz, R. Barber, ”Efficient and Effective Querying by Image Content”, Research Report, RJ 9453 (83074), Computer Science, IBM Research Division, August 1993.
- R. Barber, W. Equitz, C. Faloutsos, M. Flickner, W. Niblack, D. Petkovic, P. Yanker, ”Query by Content for Large On-Line Image Collections”, Research Report, RJ 9408 (82660), Computer Science, IBM Research Division, June 1993.
- W. Niblack, M. Flickner, ”Find Me the Pictures That Look Like This: IBMs Image Query Project”, Advanced Imaging, pp. 32-35, April 1993.

Merci, à mes parents, qui m’ont permis de faire des études,
A Thierry, merci de m’avoir accepté comme diplômant au sein du groupe vision,
A Christian, toi qui t’es tant démené pour me fournir un cadre de travail agréable: merci,
Merci à toi, Catherine, sans toi ces six mois de travail auraient été beaucoup plus fastidieux,
Et finalement, Lorna, toi qui m’a toujours affirmé que j’y arriverai, pour ton soutien, merci.

Table des Matières

1 Introduction

Avec l'apparition sur le marché de machines chaque jour de plus en plus puissantes, les ordinateurs deviennent capables de traiter des informations de plus en plus volumineuses comme les sons ou encore les images numérisés.
Ce sont ces dernières, qui nous intéressent plus particulièrement, les machines de type "ordinateur personnel" sont désormais capables de gérer des images de plusieurs millions de pixels avec un nombre de couleurs si élevé qu'il dépasse la capacité de discernement de l'œil humain.

Ainsi, avec les possibilités de traitement qu'autorise la numérisation, les images informatiques ne pouvaient que se multiplier. Il est aujourd'hui possible de trouver des images sur n'importe quel sujet, et c'est ici qu'un problème se pose. Pour des raisons qui lui sont propres, un utilisateur quelconque peut avoir un besoin urgent d'une image qui comporte une voiture rouge garée sur de l'herbe verte. Il se rappellera peut-être avoir déjà vu une image de ce genre quelque part, mais aura certainement beaucoup de mal à la retrouver sans outils spécifiques

Or, il se trouve que de tels outils sont encore introuvables sur le marché. C'est après avoir constaté ce manque que certains laboratoires de recherche se sont attelés à produire un logiciel qui soit capable de retrouver, si possible, une image en fonction de son contenu. Et parmi les produits qui sont apparus, on pourra citer le système QBIC (Querying By Image Content [4]) produit par les laboratoires de recherche d'IBM au cours de l'année 1993.

QBIC est une base de données qui gère des images et qui permet de les retrouver en fonction de leur contenu. Il offre des recherches par couleurs, par textures ou encore par contours simples. On remarquera que l'accent a été mis en particulier sur l'interface qui semble claire et agréable. Lors de leur insertion dans la base de données, les images sont précompilées et différents éléments sont calculés: une image réduite, des données directement exploitables sur les couleurs de l'image, les textures contenues dans l'image et les contours de l'image.

Notons que ce logiciel tourne sur un système RS6000 modèle 350 [14], une machine RISC qui n'a rien de commun avec un PC, aussi puissant soit-il. De plus, un tel système est "fermé", autrement dit, toute personne qui en fait l'acquisition, l'achète "en l'état" et est alors à la merci de l'éditeur. Lui seul décidera s'il veut améliorer son produit, ou en rester là.

Dans ce cas, il ne reste plus qu'une solution: faire soi-même son logiciel de base de données d'images avec recherche par contenu qui contourne ces deux problèmes, tous en utlisant des méthodes différentes. Et c'est le sujet de ce travail de diplôme (cf Enoncé) . Le produit qui en résultera devra:

- Fonctionner sur une machine très répandue. A ce niveau, le choix était assez restreint: d'un côté la dynastie Intel™ avec le compatible-PC, de l'autre le clan Motorola avec le Machintosh™. Notre choix s'est porté sur le PC, car c'est celui qui est le plus familier à l'auteur, et de toutes manières, le plus répandu.

- Utiliser une interface graphique. Ce qui est la moindre des choses pour une application destinée à traiter des images, et conviviale, ce qui est déjà moins évident. Pour cela encore, le choix a été assez limité: Windows 3.1 est le seul système qui soit à la fois répandu (il est livré en standard avec pratiquement n'importe quelle machine), qui autorise des interfaces évoluées et qui met à disposition des outils efficaces pour les créer.

- Etre programmé en C. Car la programmation windows se fait en C. On a bien assisté à quelques timides apparitions de Turbo-Pascal sur la scène de la programmation Windows, mais personne n'a semblé vraiment intéressé.

- Implémenter divers types de recherche par contenu, parmi les plus classiques: la recherche en fonction d'une couleur contenue dans l'image, et la recherche en fonction d'une texture elle aussi contenue dans l'image. De plus cette recherche devra être efficace: de l'ordre de quelques secondes. Il n'est pas question d'immobiliser une machine pendant une nuit entière pour retrouver une image.

- Le système devra être "ouvert". Autrement-dit il devra être possible de faire évoluer le produit en rajoutant de nouveaux types de recherche, et ce, en modifiant au minimum le code existant.

1.1 SPIP

La réponse au problème posé par l'énoncé est contenue dans ces quatre lettres S.P.I.P. (Search Picture In Pictures) qui est une application permettant de créer des bases de données contenant des images, pour ensuite retrouver ces dernières en fonction de leur contenu.

SPIP est capable de manipuler différents formats d'image (BMP, TIFF, GIF, SunRaster, IMA). Les images qui sont sauvée dans un de ces formats peuvent être chargées une à une pour être enregistrées dans la base ou être chargées et stockées dans la base selon un processus automatique.

Pour retrouver les images dans la base, en plus d'une consultation "simple", SPIP offre trois types de recherche:

- La recherche par nom: simple recherche en fonction des noms de fichier répertoriés dans la base de données.
- La recherche par couleurs, où les images sont recherchées en fonction des couleurs qu'elles contiennent.
- La recherche par textures, où les images sont recherchées en fonction des textures qu'elles contiennent.

Le contenu de la base de données est consultable sous forme de petites images iconifiées, répliques des images originales. Bien entendu, SPIP est capable d'afficher les images originales correspondantes, en tenant compte des particularités hardwares du matériel sur lequel il tourne.

De plus SPIP offre une aide contextuelle très complète pour aider l'utilisateur qui pourrait se trouver en difficulté. Cette aide est particulièrement détaillée en ce qui concerne les paramètres de recherche.

Et finalement, SPIP est programmé de manière à rendre les modifications ultérieures le plus simple possible: pour rajouter un type de recherche ou un format de fichier, il suffira d'ajouter un nombre réduit de lignes dans le code existant.

1.2 Rapport

Ce rapport se divise en huit chapitres, le premier étant constitué par cette introduction, les sept suivants présentent les aspects théoriques de la programmation de SPIP:

- Le chapitre 2 traite des images traitées par SPIP, tant au niveau de leur lecture sur disque qu'au niveau de leur affichage à l'écran.

- Le chapitre 3 traite les différents aspects de la base de données.

- Le chapitre 4 traite de l'aspect "ouvert" de SPIP, autrement dit des facilités qui sont offertes pour permettre à notre application d'évoluer.

- Le chapitre 5, quant à lui, décrit les différents types de recherche proposés dans la base.

- Le chapitre 6 résume les résultats obtenus avec SPIP, et fournit quelques mesures sur les performances de notre application.

- Le Chapitre 7 offre un rapide tour d’horizon quant aux possibilités d’évolution de SPIP.

- Les annexes (références bibliographiques, inventaire...) sont regroupées dans le chapitre 8.

2 Images

En même temps que les images informatiques, sont apparus les formats d'image. En effet chacun s'est empressé d'inventer le format d'image qui l'arrangeait en fonction de ses propres besoins. Le résultat ne s'est pas fait attendre: une multitude de formats différents est apparue. Heureusement, seul un petit nombre à réussi à s'imposer. Cependant une application qui prétend manipuler des images se doit d'être capable de lire un nombre raisonnable de formats, parmi les plus connus.

2.1 Formats et conversion

Il a été décidé d'implémenter 5 formats: TIFF, BMP, GIF, SUN-RASTER et IMA. Les trois premiers nommés ont été choisis parce qu'ils sont les plus utilisés actuellement [1], le choix des formats Sun-Raster et IMA a été fait pour des raisons purement pratiques.

- **Les types de données**

  Les données contenues dans un même fichier d'image d'un format spécifique peuvent être de différents types: 1, 4, 8, 24 ou 32 bits, ce qui correspond respectivement à 2, 16, 256, 16 millions ou16 millions de couleurs. Le fait que le 32 bits n'autorise que seize millions de couleurs n'est pas une erreur. Certains formats autorisent le 32 bits pour des raisons de vitesse de traitement. En effet, sur la plupart des machines, il est bien plus rapide de transférer des blocs de mémoire de 4 bytes que des blocs de 3 bytes.

  De plus, ces données peuvent être organisées suivant un modèle "ras-

ter", où chaque byte ou groupe de bytes correspond directement à un pixel de l'image originale. Cela présente l'avantage d'être simple, mais très gourmand en place. Un moyen de palier à ce problème de place consiste à compresser les données de l'image, ce qui peut être fait selon différentes méthodes, parmi lesquelles on distingue:

RLE (Run Length Encoding)

Ce type de compression est basé sur une constatation selon laquelle un document contient souvent des suites de bytes similaires. En effet, pour coder une ligne de 255 points noirs qui se suivent, il est beaucoup plus rentable de stocker le couple {255,0} plutôt que 255 fois la valeur 0. Ce type de compression est assez efficace sur des images qui contiennent de grandes zones constantes, et permet des taux de compression qui varient de 2 pour 1 à 5 pour 1 sur ce type d'image [2]. Cependant cette méthode ne donne pas des résultats assez satisfaisants pour des images dites "naturelles", ainsi que pour celles qui sont codées en 24 ou 32 bits[1].

LZW (Lempel-Ziv and Welch)

Il arrive assez fréquemment que l'on rencontre plusieurs fois la même suite de bytes dans une image, il est donc possible de gagner de la place en codant cette suite de manière plus compacte. La compression LZW, basée sur ce principe, donne des résultats assez remarquables, en particulier sur les images trammées, avec des taux de compression pouvant aller jusqu'à 16 pour 1 [2]. Il pourrait cependant être intéressant de noter que l'algorithme LZW est protégée par un copyright, dont peu de gens semblent faire cas.

Chaque format peut en général supporter plusieurs types de données et de compressions. Les formats qui ont été implémentés sont:

BMP (BitMaP)

Il s'agit du format "officiel" Windows, et le seul que ce système comprenne. Le format BMP est capable de supporter 1, 4, 8 et 24 bits, et permet d'organiser les données en format raster non-compressé ou avec compression RLE. Cependant cette dernière possibilité n'est presque jamais exploitée. Pour des raisons de simplicité, tous les formats chargés par l'application sont convertis de manière interne au format BMP 24 bits non compressé. Le 24 bits non-compressé est le seul qui autorise n'importe quel type de traitement d'images, le choix de BMP, quant à lui, est dû à Windows.

TIFF (Tag Image File Format)

Ce format est censé avoir une vocation universelle, il permet quantité de formats internes et de compressions, ce qui rend son décodage assez malaisé [2].

GIF (Graphics Interchange Format)

Créé par CompuServe Incorporated™, ce format n'admet que le 1,4 et 8

1. Le fait de coder les points d'une image 24 bits sous forme d'un triplet (R,G,B) détruit toute continuité.

bits en compression LZW uniquement. Si ce n'est pas le plus utilisé, il s'agit certainement du plus connu [2].

SUN-RASTER

C'est le format officiel de SunMicroSystems™, qui autorise les rasters 8,16, et 32 bits compressés ou non (RLE). Le fait qu'il n'existe pas de documentation "officielle" de la part de Sun, a incité certains à lui apporter quelques modifications, à savoir: les fichiers Sun-raster, générés sur un PC, ne sont pas exactement les mêmes que ceux générés sur une station Sun.

IMA

Il s'agit d'un format utilisé par le logiciel "LaboImage"[1] qui présente la particularité d'être facile à décoder. Son entête est stockée dans un fichier texte tandis que ses données sont stockées dans un fichier binaire séparé. Il n'autorise aucune compression interne [18].

Des informations complémentaires, sur ces formats d'images ainsi que sur les différents algorithmes de compression / décompression, peuvent être trouvées dans [2].

La Table 1: offre un récapitulatif des formats et sous-formats supportés par notre application. Un "oui" signifie que l'option spécifiée est autorisée par le format et implémentée, un "non" veut dire qu'elle est autorisée par le format, mais qu'elle n'est pas supportée par SPIP, enfin un "-" signifie que l'option n'est pas autorisée par le format.

	BMP	TIFF	GIF	Sun-Raster	IMA
1 bit / pixel	oui	oui	oui	oui	non
4 bits / pixel	oui	oui	oui	oui	-
8 bits / pixel	oui	oui	oui	oui	oui
16 bits / pixel	-	-	-	-	oui
24 bits/ pixel	oui	oui	-	oui	oui
32 bits/ pixel	-	-	-	oui	-
complexe	-	-	-	-	non
multi-plan	-	non	-	-	oui
LZW	-	oui	oui	-	-
RLE/ Packbit	non	-	-	oui	-

Table 1: : formats acceptés par SPIP

1. LaboImage est un logiciel de traitement d'image, développé par le groupe vision, sous la direction du Pr. Pun, distribué gratuitement aux universitée qui en font la demande..

Une fois passée dans un format unique, à savoir BMP 24 bits, l'image chargée peut être traitée. Il y a en particulier deux opérations qui vont s'avérer être des points de passage obligé. Ces opérations sont décrites au paragraphe suivant.

2.2 Réductions d'images

Il n'est en effet pas concevable d'imaginer, pour une base de données d'images, une interface qui ne permettrait de manipuler les images que sous forme de noms de fichier, pas plus qu'il n'est raisonnable de vouloir manipuler les images entières sous leur forme originale. Elles prendraient beaucoup trop de place tant en mémoire que sur l'écran. Il est donc judicieux de calculer une réduction de chacune des images lors de leur insertion dans la base de données:

Une taille carrée de 100 pixels de côté a paru raisonnable[1] pour ces images réduites, cependant un rapide calcul nous indique qu'une image carrée de 100 pixels de côté, codée sur 24 bits occupe un peu moins de 30Ko. Par conséquent une base de données, qui contiendrait 1000 images, occuperait au minimum 30Mo, ce qui n'est absolument pas concevable.

La solution, qui vient tout de suite à l'esprit, est de réduire encore un peu les dimensions de l'image, mais une image trop petite est inutile, car inexploitable. Donc, si l'on ne peut réduire davantage les dimensions, il ne reste plus qu'à réduire la profondeur et passer de 24 bits à 4. Cela nous permet de gagner un facteur 6 et ramène la taille de notre base à 5 Mo.

Il faut donc convertir une image 24 bits de dimension quelconque (largeur, hauteur et profondeur) à une taille maximale de 100x100 pixels avec une profondeur de 4 bits.

Réduction

La réduction de taille peut se faire de la manière suivante. Soit une image de dimensions LxH constituée de trois plans $R1_{i,j}$, $G1_{i,j}$, $B1_{i,j}$, $i = 0..(L-1), J=0..(H-1)$

On calcule tout d'abord le coefficient de réduction C, qui est le centième de la plus grande dimension de l'image; ainsi, l'aspect ratio[2] de l'image sera conservé. Notons que le coefficient de réduction ne peut être inférieur à 1: on ne cherche pas à agrandir l'image, mais à la réduire.

$$C = max\left(1, \frac{max(H, L)}{100}\right)$$

On partitionne ensuite l'image originale en clusters de taille CxC et on calcule les composantes de $R2_{i,j}$, $G2_{i,j}$, $B2_{i,j}$, $i = 0..(L-1)/C$, $J=0..(H-1)/C$

1. Avec une taille de 100x100 pixels, on fait tenir environ 25 images à la fois dans une fenètre qui occupe le quart d'un écran 1024x768. Une taille plus petite rend les images moins claires.
2. Déf: l'aspect ratio d'une image est le rapport largeur sur hauteur.

de chaque pixel de l'image réduite comme étant la moyenne des composantes *R1,G1,B1* des pixels se trouvant dans le cluster correspondant.

$$R_{i,j} = \frac{1}{C^2} \sum_{j=jC}^{(j+1)C} \left(\sum_{k=iC}^{(i+1)C} R_{k,l} \right)$$

Considérations analogues pour $G_{i,j}$ et $B_{i,j}$

Ce moyennage permet d'éviter certains phénomènes de sous-échantillonage [10], en particulier sur les quadrillages (c.f. fig 1, b et c): il se comporte comme un filtre passe-bas simple.

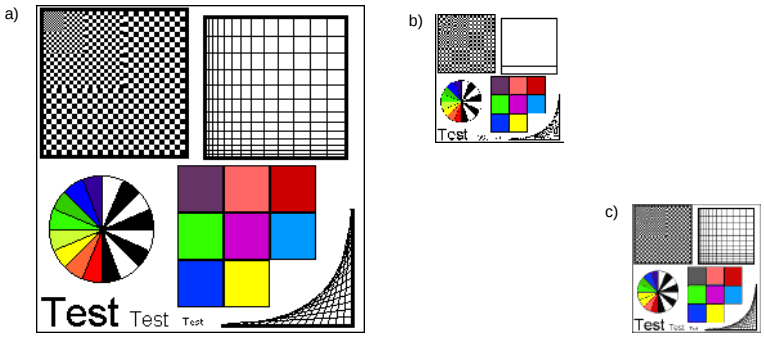

fig 1 : Réduction d'images:
a) image à réduire.
b) image réduite sans moyennage.
c) image réduite avec moyennage.

La requantification à 4 bits

La requantification quant à elle risque de s'avérer un peu plus complexe. Lorsqu'il s'agit de passer de 16 millions à 16 couleurs il faut retrouver les seize couleurs les plus représentées dans l'image car une palette de 16 couleurs est trop réduite pour faire du tramage (c.f. 2.3). Cela peut se faire à l'aide d'un histogramme à trois dimensions dont les trois axes sont représentés par les composantes R, G et B des couleurs. Chaque axe doit être gradué de 0 à 255, mais l'on se rend rapidement compte que, si chaque cellule de l'histogramme est codée sur 4 octets[1], le tout prendra 64 Mo, ce qui ne tiendrait même pas dans la mémoire vive de la machine. Il faut donc partitionner chacun des axes de notre histogramme en 16 parties par exemple, ce qui va réduire notre histogramme à 16Ko.

En principe, il suffirait de trouver les 16 clusters (cubes) les plus remplis pour avoir les 16 couleurs principales; mais il s'agit là d'une très mauvaise approche. En effet, dans une image, il peut y avoir beaucoup de nuances de la même couleur (dans les dégradés en particulier) qui risquent d'être comptabilisées non seulement comme des couleurs différentes mais

1. Ceci est un minimum: 2 bytes ne permettraient pas de dénombrer plus de 65535 pixels de la même couleur dans une image, or la plupart des images en contiennent beaucoup plus que cela.

aussi très représentatives de l'image, et ce, au détriment d'autres couleurs moins représentées mais néanmoins importantes. Un cas typique serait un cercle rouge sur un fond dégradé gris (c.f. fig 2). La couleur rouge du cercle serait remplacée par un des niveaux de gris du dégradé.

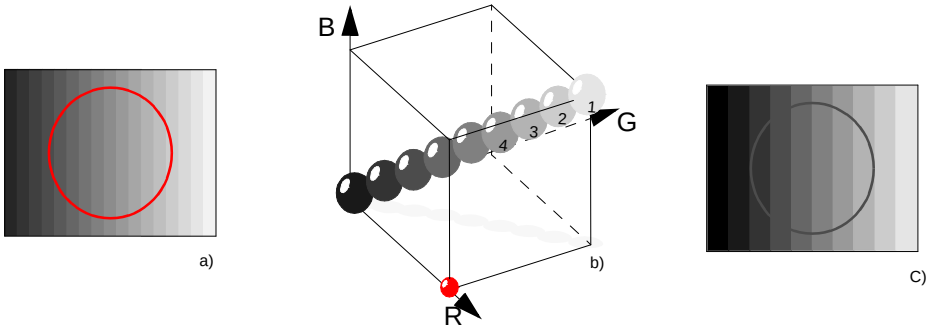

fig 2 : Problème de requantification:
a) image à requantifier.
b) histogramme correspondant.
c) image requantifiée..

Ce problème peut être résolu en imposant une distance minimale entre deux cluster retenus. On recherche la couleur la plus représentée (c.f. fig 2b.1), puis la deuxième couleur la plus représentée (c.f. fig 2b.2). Si elle est trop proche de la première (clusters adjaçants) elle n'est pas retenue, et ainsi de suite jusqu'à avoir 16 couleurs.

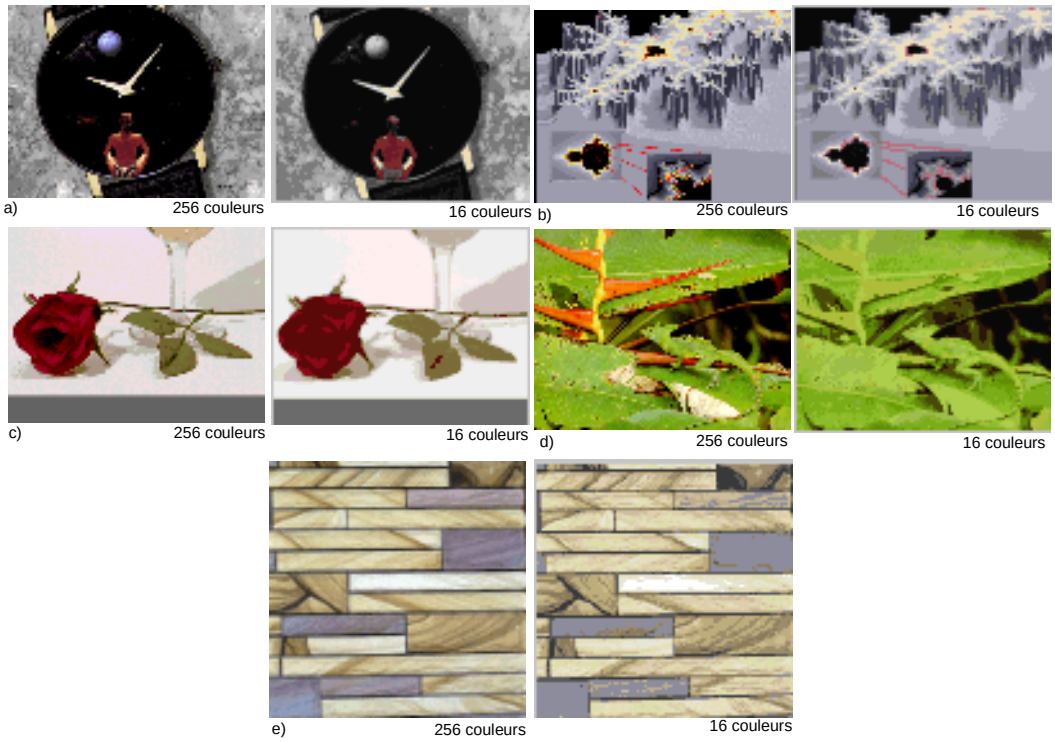

fig 3 : la requantification par dénombrement des couleurs (256 -> 16).

On remarquera dans les images de la fig 3 que, si dans l'ensemble les résultats sont satisfaisants, l'image d), qui représente un petit lézard sur une feuille, a eu quelques problèmes lors de la requantification, car le vert a malgré tout accaparé la quasi-totalité de la palette.

2.3 Requantification

Même si des cartes graphiques capables de gérer le 24 bits à un prix abordable commencent à apparaître sur le marché, la plupart des cartes vidéo actuelles ne sont pas capables d'afficher plus de 256 couleurs parmi 16 millions. Cela qui implique l'utilisation de tables de look-up, plus connues sous le nom de palettes de couleurs.

On va donc se trouver confronté à un premier problème. Comment, lors de la réduction de 24 à 8 bits , calculer une palette qui autorise un rendu optimal?

• Comptabiliser les couleurs

Une solution consiste à opérer de la même manière que pour la requantification à 4 bits (c.f. 2.2). Cependant, outre le fait que le temps de calcul de la palette croît de manière exponentielle[1] fonction de la taille de la palette à calculer, on se trouve confronté à un problème lors de l'affichage de l'image dont on est en train de calculer la palette.

Lorqu'on aura calculé la palette optimale, on affiche notre image, tout se passe bien. Mais lorsque l'on voudra afficher une deuxième image, sans pour autant effacer la première, un conflit de palettes se produira (c.f. fig 4).

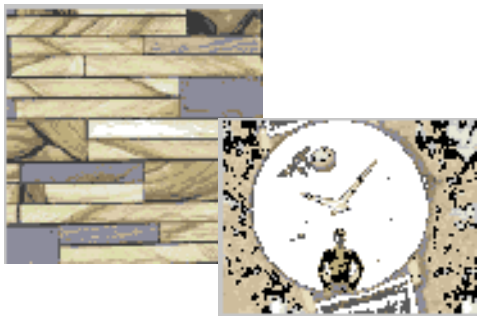

fig 4 : un conflit de palette.

En effet, la première image ayant réservé les 256 couleurs de la palette de couleurs du système, il n'en restera plus pour la deuxième. Dans le meilleur des cas, le système d'affichage s'en apercevra et tentera de trouver les couleurs qui correspondent le mieux dans la palette actuelle, mais en général le résultat est souvent discutable. Dans le pire des cas, l'affichage se fera directement avec la palette en cours.

On peut donc en conclure que pour afficher des images couleurs, calculer une palette de couleurs optimale n'est pas la bonne solution.

1. Plus il y a couleurs à trouver dans la palette, plus il y a de distances minimales (telles celles décrites au pragaraphe 2.2) à contrôler entre les couleurs retenues. Cela n'est gère important lorsque l'on cherche 16 couleurs, mais le devient quand il y en a 256.

• Tramer les images

Imaginons une image qui soit uniquement constituée de points rouges et bleus disposés en damier. Grâce à l'effet intégrateur de l'oeil humain, cette image n'apparaîtra pas comme un damier mais comme un rectangle violet, dont les composantes (R,G,B) sont les moyennes respectives des composantes du rouge et du bleu.

En creusant cette idée, on comprend que l'on peut assez aisément obtenir une pseudo couleur C_0 de coordonnées (R_0,G_0,B_0) en disposant de manière judicieuse des couleurs issues d'une palette fixe dans un carré de 4x4 pixels que l'on appellera le motif de la trame.

Sur ce principe, on peut remplacer chaque pixel de l'image originale par le carré correspondant. Cependant, si on applique cette méthode ainsi, on se rend bien compte que l'image calculée de la sorte va voir ses dimensions multipliées par 4, ce qui est inacceptable. On peut alors résoudre le problème en donnant à chacun des pixels de l'image la couleur de son vis-à-vis dans un image fictive construite avec le motif de la trame répété (c.f. fig 5).
Ce qui, sous une forme un peu moins poétique, donne:

$$R_{i,j} = T_{i|n,j|n}(I_{i,j})$$

Où $R_{i,j}$ est le pixel de coordonées (i,j) dans d'image à calculer; $I_{i,j}$ le pixel de l'image originale de coordonnées (i,j) et $T_{i,j}(c)$ l'élément de coordonnées (i,j) de la matrice de génération de texture de taille nxn qui correspond à la couleur c. R, T et I sont des triplets (R,G,B). $a|b$ signifie la division modulo.

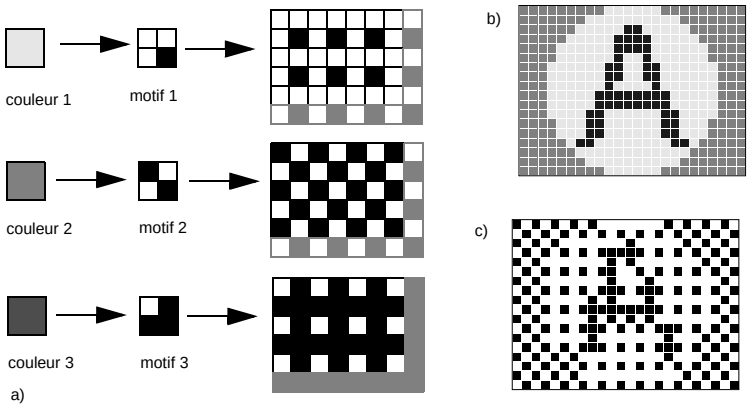

fig 5 : Un exemple simple de trammage avec trois couleurs de base:
a) tramage des couleurs 1,2 et 3.
b) image originale, avec les couleurs 1,2 et 3.
c) image tramée.

Il faut cependant bien prendre conscience que le résultat ne sera acceptable qu'à des résolution importantes, 800x600 sur un écran 14 pou-

ces au minimum.
Mais comme il est possible de le constater (c.f. fig 6), le résultat est assez convainquant: il s'agit pourtant d'une image qui contient beaucoup de dégradés, qui sont en général assez réfractaires au tramage.

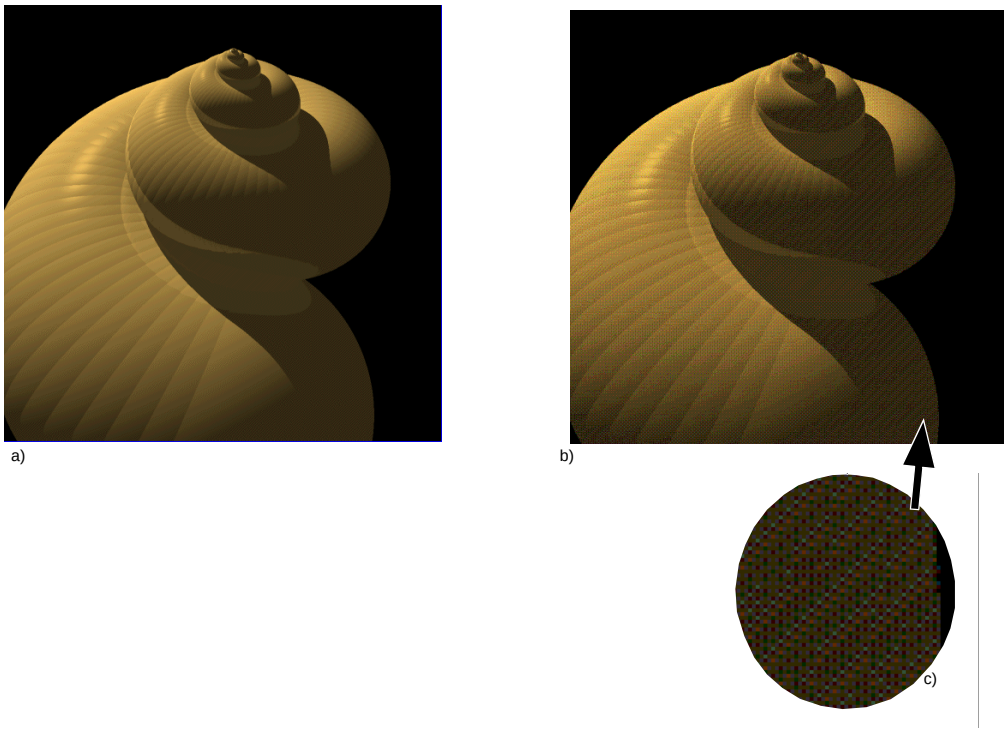

fig 6 : Tramage: résultats:
a) une image normale (24 bits).
b) la même, tramée.
c) détail de l'image b.

Notons par ailleurs que si une carte vidéo permet d'afficher directement 16 millions de couleurs sur l'écran, le tramage devient non seulement inutile, mais indésirable en raison de la perte de qualité qu'il entraîne. C'est pourquoi les deux types d'affichage (tramé et non tramé) doivent coexister, notre application devant tourner sur les deux types de configuration.

3 DataBase

Comme il l'a déjà été dit précédemment, le système constitué par la base de données doit être ouvert, ce qui signifie qu'il doit être possible d'y ajouter des modules, sans devoir passer trop de temps à analyser le code existant. C'est pourquoi SPIP a été divisé en trois parties indépendantes.

3.1 Architecture: trois blocs distincts

On distingue donc:

- **Interface Windows et Gestion bas niveau de la base**

  Il s'agit de la programmation de l'interface principale de Windows, ainsi que de toute sa logique. Elle comprend aussi toutes les fonctionnalités de la base de données telles que "ajouter/supprimer" des images, ou encore accéder à une image particulière. C'est en fait la partie principale de l'application, et c'est elle qui contrôle les deux autres.

- **Formats de fichiers d'images**

  Il est important de rendre l'application la plus indépendante possible de la multitude de formats d'images existants, ce pour deux raisons: d'une part cela rend la programmation beaucoup plus aisée, d'autre part cela facilite grandement l'ajout d'autres formats dans la base. C'est pourquoi cette partie a été divisée en deux couches: la première qui comprend les modules qui gèrent explicitement les images codées dans les formats reconnus (reconnaissance, chargement, information). La deuxième qui se charge de coordonner ces modules tout en proposant un format unique (BMP 24bits) aux couches supérieures (c.f. fig 7).

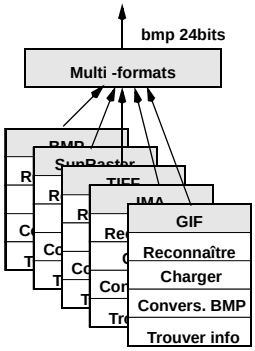

fig 7 : Couche "Formats de fichiers".

On comprend donc qu'il est assez facile d'ajouter des fonctionnalités qui permettent de reconnaître et de charger des formats d'images supplémentaires: il suffit de rajouter quelques lignes dans le fichier qui contient le code de la couche "multi-format".

• Paramètres de recherche

Les modules de recherche sont aussi du code qu'il faut pouvoir ajouter très facilement, de la même manière que pour les formats de fichier, mais cette fois le formalisme a été poussé un peu plus loin. En effet, lorsque l'on implémente un nouveau type de recherche, on est tenu de coder un certain nombre de fonctions officielles (c.f. fig 8), chacune devant effectuer une tâche bien précise. L'une d'entre elles est chargée de renvoyer au code de la base les adresses de toutes les autres. De cette manière, il suffit de rajouter une seule ligne de code (dans le code existant) pour rajouter un paramètre de recherche. Ce qui, comme on peut le constater, est très simple.

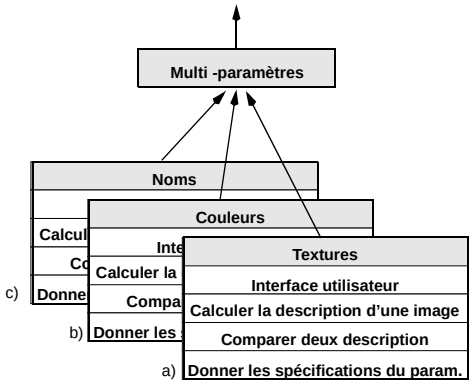

fig 8 : Couche "paramètres de recherche":
a) module "recherche par texture".
b) module "recherche par couleur".
c) module "recherche par nom".

Nous rediscuterons plus en détail de ces fonctions au paragraphe 4.3.

3.2 La base de données

Ce paragraphe traite des données qui sont stockées dans la base.

• Les images

Comme on l'a déja vu, il n'est absolument pas question de stocker les images entières: cela prendrait beaucoup trop de place, il est bien plus raisonnable de stocker un lien sur chaque image, c'est-à-dire une simple chaîne de caractères. Mais il faut alors noter que les images référencées ne seront pas toujours disponibles. Elles peuvent en effet être stockées sur un CD-ROM qui ne se trouvera pas forcément dans le lecteur lors de la consultation.

Il est évident que le contenu de la base de données devra être manipulé par l'utilisateur sous une forme visuelle, c'est pourquoi il sera judicieux de calculer une réduction pour chacune des images qui sera stockée dans la base. Comme on l'a déja mentionné les images réduites font au maximum 100x100 pixels de côté en 16 couleurs ce qui prend à peu près 5Ko par image (c.f. 2.2, page 16).

• Les datagrammes

Lors de la précompilation d'une image, chaque bloc "paramètres de recherche" (c.f. fig 8) calcule un bloc de données qui correspond à sa propre "interprétation" de l'image. Par exempl:e le bloc "recherche par couleurs" calculera un histogramme 3D. Concrètement, ce bloc est constitué d'une suite de bytes que l'on nommera dorénavant "datagramme".

La notion de datagrammes

Pour une application qui dispose de n paramètres de recherche, à chaque image stockée dans la base, correspondent n blocs de données anonymes, les *datagrammes*, qui seront enregistrés tel quels dans la base, accompagnés d'un numéro d'identification ID. Ce numéro permettra de retrouver le module de recherche qui a généré chaque datagramme.

Les datagrammes sont des données "anonymes" car la base de données n'a en effet pas besoin de connaître leur signification (i.e comment ils sont interprétés par les modules de recherche). Pour que notre application reste évolutive, le code qui gère la base de données ne doit jamais accéder aux données encapsulées dans les datagrammes. La nature de ces données peut en effet très facilement changer suivant la façon dont sont implémentés les modules de recherche.

Si l'on poursuit ce raisonnement, on s'aperçoit que cette notion de datagramme peut s'étendre aux images réduites. En effet, ces réductions n'ont de signification que pour la partie interface de la base de donnée. Et cela simplifie les choses: il n'y aura pas besoin de programmer explicitement une fonction qui enregistre puis récupère les réductions, puisque l'on pourra se servir de celles qui gèrent les datagrammes.

On peut donc esquisser assez aisément l'allure d'un élément de la base de données (c.f. fig 9). Il sera constitué:

- D'une entête, ou descripteur, qui aura pour tâche de spécifier le lien avec l'image originale, d'adresse du prochain descripteur dans le fichier de la base,
- Une table des descripteurs qui permettra de retrouver l'adresse dans le fichier, la taille et l'ID de chaque datagramme,
- Et enfin, des datagrammes qui seront stockés les uns après les autres.

fig 9 : Un élément de la base.

Normalement ce fichier devrait amplement suffire au code qui gère la base de données. En effet, il est possible d'accéder à n'importe quel datagramme à partir du début du fichier, en utilisant les liens qui permettent de sauter d'un descripteur à son suivant, cependant on n'a aucune garantie sur le temps que cela mettra. Pour accèder au $n^{ième}$ datagramme, il faut en effet parcourir les n-1 descripteurs qui le précèdent et ainsi faire n accès disques. Pour pallier à cet inconvénient, on crée un fichier d'index qui contient, à la suite, les adresses de tous les descripteurs stockés dans la base. Le prix à payer pour cette efficacité accrue (plus que 2 accès disques, pour accèder au $n^{ième}$ datagramme[1]) est l'obligation de tenir à jour ce fichier d'index, notamment lors de la suppression d'images de la base.

Par ailleurs, ce type de structure, utilisant une table des datagrammes, offre une compatibilité accrue lors de l'ajout de modules de recherche dans la base. Une base contenant des datagrammes issus de modules de recherche A, B et C sera compatible avec une application disposant des modules de recherche A, B, C et D; les datagrammes de type D ne seront tout simplement pas disponibles (i.e. la fonction de distance renvoie la valeur maximale) lors d'une recherche selon D.

1. Un premier accès au fichier d'index pour lire l'adresse du datagramme, et un deuxième accès à la base pour lire le datagramme. Notons que lorsque l'on doit accéder à plusieurs descripteurs, on peut se contenter de charger une seule fois le fichier d'index en mémoire vive (ce que fait SPIP).

La dualité datagramme utilisateur / image

Nous disposons d'une base de données qui comporte une description de chaque image par l'intermédiaire des datagrammes. Il reste maintenant à faire une recherche à l'intérieur de cette base, et pour cela, il est nécessaire de connaitre le paramètre qui va être utilisé. Par exemple: "toutes les images qui contiennent du rouge". Ce paramètre doit être spécifié par l'utilisateur par l'intermédiaire d'une interface qui, en raison de sa spécificité, appartient au module de recherche. Cette interface se devra de renvoyer un bloc de données que l'on nommera *datagramme utilisateur* par opposition à *datagramme image* (celui qui est stocké dans la base et que l'on a décrit au paragraphe précédant). Notons que ce datagramme image, doit être, lui aussi, traité comme une boîte noire par la partie de l'application qui assure la gestion de la base de données.

La recherche à l'aide des datagrammes

Lors de la recherche, il ne reste plus qu'à récupérer chaque datagramme image et à demander au module de recherche correspondant de calculer la distance entre ce datagramme image et le datagramme utilisateur obtenu via l'interface (c.f. fig 10). On ne conservera en fin de compte que les références des descripteurs qui auront permis d'obtenir les n plus petites distances, ($n = 32$ pour spip[1]) et on utilise ces références pour afficher le résultat de la recherche.

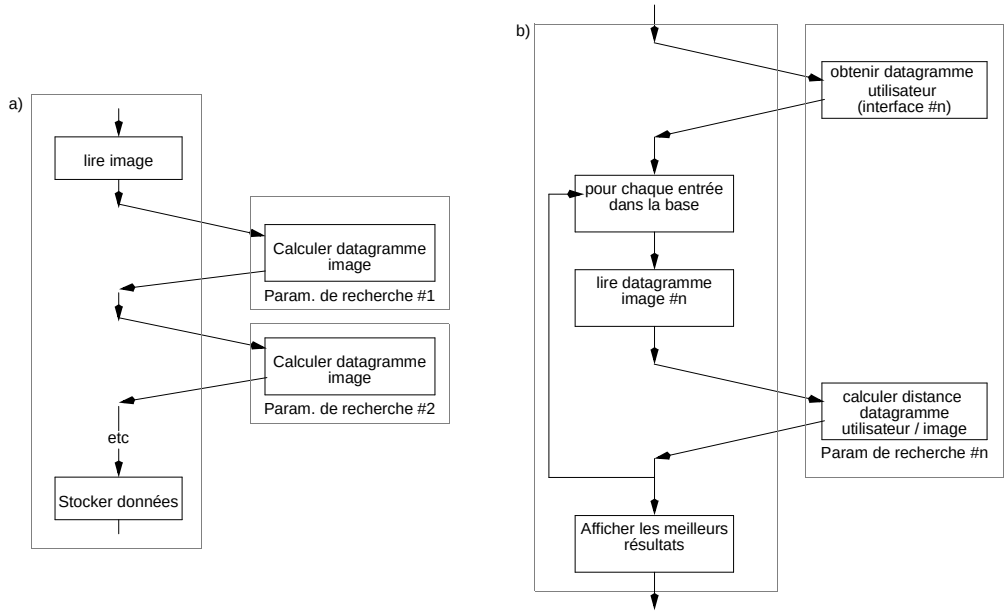

fig 10 : Ajouter et rechercher une image dans la base:
a) ajouter une image.
b) rechercher une image selon le paramètre de recherche #n.

1. 20 images pour QBIC [4]

- **Les recherches combinées**

Il est possible que l'on désire, non pas effectuer une simple recherche, mais retrouver une image qui corresponde à plusieurs paramètres de recherche. Ainsi, pour chaque entrée dans la base, il faudra faire appel à tous les modules "paramètres de recherche" et combiner les résultats.

Afin de ne pas favoriser un type de recherche par rapport aux autres, il faut que les distances renvoyées par chaque module "paramètre de recherche" soient toutes du même ordre de grandeur, c'est pourquoi les valeurs renvoyées par les fonctions de distances de ces modules doivent être normalisées (entre 0 et 1, c.f. chap 5).

Par ailleurs, on peut avoir envie de favoriser volontairement un type de recherche. Par exemple, si l'on cherche une image qui représentait une voiture rouge dont le nom de fichier serait "??car??", on fera une recherche qui porte sur les noms de fichiers qui contiennent les trois lettres "car" et les images qui contiennent une couleur "rouge" en accordant beaucoup plus d'importance à la couleur. Pour ce faire, il suffit d'associer un coefficient à chaque paramètre de recherche lors de la combinaison.

La distance finale sera donc:

$$R_i = \sum_j C_j D_{i,j}$$

Où R_i, est le résultat pour l'image i, $D_{i,j}$ la distance renvoyée par la fonction de distance du module de recherche j, et C_j le coefficient accordé au paramètre j.

SPIP gère ce problème (favoriser ou non certaines recherches), accordant autant d'importance à chaque paramètre lors de la recherche (i.e. $C_j = 1$), mais permet ensuite de modifier l'ordre des résultats de cette dernière en modifiant les coefficients C_j grâce à des curseurs.

Par ailleurs, l'interface de SPIP permet d'afficher les résultats numériques (i.e les distances calculées) d'une recherche.

3.3 Fonctionnalités essentielles

En pratique, une base de données doit être capable de fournir un certain nombre de fonctions de "standards".

- **Les fonctionnalités "ordinaires"**

Tout d'abord on distingue les fonctions les plus évidentes telles que:

ajouter une image

C'est bien évidemment la fonction la plus importante. Bien que certains systèmes tels que QBIC [4] ne permettent pas d'ajouter facilement des

images dans la base, à moins de la reconstruire entièrement. Il a paru indispensable de permettre à notre application d'effectuer ces ajouts facilement par l'utilisateur.

Lors de l'ajout d'une image, on distingue deux cas de figure:

- Rajouter une seule image.
- Rajouter plusieurs (beaucoup) images dans la base à la fois.

En effet, si l'utilisateur désire entrer la totalité du contenu d'un CD-ROM, il ne peut se permettre de spéficier le nom de chaque image, une par une: cela représente beaucoup trop de travail.

- Rajouter une seule image:
  L'utilisateur peut avoir envie de ne rajouter qu'une seule image, auquel cas le problème est assez simple: il lui suffit de spécifier le nom du fichier concerné, l'application n'a plus qu'à lire l'image, la compiler (i.e calculer les datagrammes) et la rajouter dans la base.

- Rajouter plusieurs images:
  C'est l'application qui trouve les noms de fichiers en parcourant un répertoire spécifié par l'utilisateur. Puis chaque image est compilée et insérée dans la base de données. Notons que l'application ne peut se baser sur les extensions des noms de fichiers pour reconnaitre ceux qui contiennent des images; en effet, nombre d'éditeurs de CD-ROM se permettent d'utiliser des extensions non-standard.

  Par ailleurs, afin de rendre ce système d'ajout automatique, plusieurs options devront être disponibles:

  - Reconstruire la base.
  - Autoriser ou non les fichiers possédant le même nom.
  - Ne rajouter que certaines images (en fonction de leur format).

  - Reconstruction
  Il doit bien sûr être possible de décider de reconstruire la base à partir de rien, autrement dit d'effacer l'ancienne base et d'en recommencer une nouvelle.

  - Ajout d'images ayant le même nom.
  Lors de l'ajout automatique d'image, on se retrouve devant un choix assez problématique. Comment réagir lorsqu'une image possède le même nom de fichier qu'une image qui est déjà présente dans la base? Ici plusieurs cas peuvent se présenter.

  Le fichier existant dans la base porte le même nom, la même extension, et est accessible par le même chemin, auquel cas la question ne se pose pas: on est en présence du même fichier, il est inutile de le rajouter.

Le fichier porte le même nom, mais pas la même extension, ou n'est pas accessible par le même chemin: cela arrive fréquemment avec les CD-ROM où les éditeurs ont tendance à mettre chaque image en plusieurs exemplaires mais sous des formats différents. Dans ce cas, la parade consiste à proposer à l'utilisateur une option, qui autorise ou non l'ajout des images qui entrent dans ce cas de figure.

Une autre façon de gérer le problème des noms de fichiers identiques consiste à permettre à l'utilisateur de n'autoriser que certains formats de fichiers (par exemple seulement les fichiers BMP). Rappelons ici que notre application ne doit pas se servir des extensions de fichiers pour retrouver les images correspondantes.

Notons qu'au niveau de l'implémentation pure, l'ajout d'images peut s'effectuer de plusieurs manières:

- La première qui vient à l'esprit consiste à ouvrir la base et à rajouter les nouveaux descripteurs à la fin. Mais un certain nombre d'incidents, laissés à l'imagination du lecteur, peuvent se produire pendant la compilation des images, qui peut prendre dans certains cas plusieurs heures. Ainsi suivant les cas, la base risque d'être endommagée.
- Une solution plus sûre consiste à copier la base de données dans un fichier temporaire, puis ensuite à mettre à jour ce fichier temporaire, et, finalement remplacer l'ancienne base par le fichier temporaire. Ainsi, en cas de problèmes graves, l'ancienne base subsistera, et l'utilisateur n'aura pas tout perdu. Notons que le prix à payer d'une telle astuce est que la mise à jour de la base demande un peu plus du double du volume de la base.

Suppression d'images dans la base.

La suppression d'images est aussi une fonctionnalité importante. En théorie, il s'agit d'une opération assez simple: il suffit de supprimer l'enregistrement correspondant dans la base.

Mais cela va laisser des trous dans la base de données et il faudra les supprimer en effectuant un décalage des données de la base. Il s'agit d'une opération typiquement longue, surtout s'il faut la faire après chaque suppression.

Le problème a été résolu en implémentant dans les descripteurs un flag de validité. La suppression d'une image consiste simplement à activer ce flag, ce qui signifie que l'image correspondante n'est en principe plus disponible. La suppression effective n'a lieu que lors de la fermeture de la session (i.e. la fermeture de la dernière fenêtre de consultation). Cette méthode présente un avantage supplémentaire: il est possible de demander en fin de session à l'utilisateur s'il désire vraiment que les suppressions qu'il a effectuées soient prises en compte. Dans l'affirmative, on supprime effectivement les enregistrements qui ont été invalidés, dans le cas contraire, il n'y a qu'à désactiver tous les flags.

- **Accès multiples**

  Il est intéressant de pouvoir consulter la base plusieurs fois en simultané. Un cas typique se présente lorsque l'on désire, à la fois, faire des recherches dans la base, et aussi avoir une fenêtre de consultation ouverte. De plus, les modifications apportées à la base dans une fenêtre de consultation doivent se répercuter dans les autres. Ainsi la suppression d'une image dans une fenêtre force les autres fenêtres à indiquer que l'image supprimée n'est plus disponible, en la grisant par exemple. De plus, il n'est pas question d'entreprendre des modifications importantes de la base de données, telles que des ajouts, pendant qu'une consultation est en cours. C'est donc pour ces raisons qu'un contrôle des accès a été instauré. Il est totalement invisible pour l'utilisateur, mais cependant il entre en action à chaque fois qu'un accès à la base est effectué.

4 Un système ouvert

Comme on l'a brièvement vu précédemment, notre application se doit d'être un système ouvert, autrement dit, un système dans lequel il sera, par la suite, très facile d'ajouter du code pour permettre une évolution. Ces facilités peuvent être présentes à plusieurs niveaux.

4.1 Les ressources

Les ressources constituent la possibilité la plus abordable pour faciliter l'évolution d'une application. C'est en effet un système qui, directement intégré à Windows, permet de définir l'aspect extérieur d'une application. Il est possible de charger un fichier exécutable, d'en modifier les ressources et de le sauver. Et cela sans même avoir à le recompiler.

- **Interface**

  L'interface de notre application est entièrement définie dans les ressources. Le code est écrit de manière à être dépendant de la géométrie de l'interface. Ainsi, lorsqu'une fenêtre d'affichage pour les images réduites apparaît, le nombre d'images affichables est automatiquement déterminé en fonction de la taille de celles-ci. Notons que la taille des fenêtres dépend directement du type de caractères utilisés par Windows pour son affichage, et par conséquent, n'est pas sous le contrôle du programmeur.

- **Les messages affichés**

  Tous les messages (i.e. chaînes de caractères) utilisés dans SPIP, sont

définis dans les ressources, ce qui permet un changement aisé de langage.

On comprendra donc que tout l'aspect extérieur de l'application est uniquement défini dans les ressources, et peut être très aisément modifié même par quelqu'un qui n'est pas familier avec la programmation Windows, voir la programmation tout court.

4.2 Rajouter des formats d'images

Les formats d'images se trouvent à un niveau où il est impératif de prévoir des facilités d'évolution en vue de mises à jour ultérieures.

- **Indépendance du format d'origine**

  Comme on l'a vu, pour limiter l'impact qu'aurait la diversité des formats d'images, il est indispensable de lire et de convertir les images le plus bas possible dans les couches de l'application (c.f. 3.1, page 23). Pour ce faire, chaque format d'image est encapsulé dans un fichier (module) différent. Chacun de ces modules est tenu de fournir un certain nombre de fonctions que nous qualifierons de "standard".

LoadXXXFile

C'est la fonction principale, qui a pour mission de lire un fichier du format XXX, et de le restituer en mémoire sous un format unique bien défini, ici le BMP 24bits.

GetXXXSpec

Où XXX est le nom du format, qui nous permet de connaître les caractéristiques principales d'un format d'images. En particulier son nom, ainsi que l'extension qu'il utilise en général.

AuthorizeXXX & isXXXAuthorized

Ces deux routines ont aussi un lien avec l'autorisation de charger certains formats d'images lors de la mise à jour automatique de la base (c.f. 3.3, page 28). Il peut paraître étrange de demander aux formats eux-mêmes de gérer leur propre autorisation, normalement de telles opérations devraient être gérées par le module de mise à jour. Un tel arrangement permet d'éviter d'avoir à maintenir en permanence une liste des formats "autorisés" tout au long de l'application.

IsAValidXXXFile

Qui permet de savoir si un fichier est effectivement un fichier du format XXX. Dans la plupart des cas, ceci est assez facile à déterminer: la quasitotalité des fichiers d'images comportent en tout début de fichier une série d'octets qui sert de signature. Rappelons, une fois encore, que faire confiance aux extensions de noms de fichiers pour identifier un format est très aléatoire.
Notons que, malheureusement, certains formats marginaux ne comportent pas de signatures, dans ce cas il faut se baser sur

d'autres caractéristiques. En l'occurrence, pour les formats IMA, les images sont toujours divisées en deux fichiers:

Un premier fichier de description portant l'extension ".DES" qui ne doit comporter que des caractères ASCII et un second qui contient les données de l'image elle-même.

Avec ces indications il est possible de reconnaître une image au format IMA. Mais bien entendu, ce processus de reconnaissance sera beaucoup plus long que lorsqu'il suffit de contrôler une simple signature.

• La couche supérieure

Tous les modules qui gèrent les différents formats d'images, sont dominés par une petite couche (c.f. fig 11,b) dont le principal but sera d'uniformiser les formats d'image. Elle comprend une fonction principale, *LoadImage*, dont la tâche est de soumettre le fichier à charger à chacune des fonctions "*IsAValidXXXFile*" de chaque module de la couche inférieure. Et si l'une d'entre elles répond par l'affirmative, de demander à la fonction "*LoadXXXFile*" correspondante de charger l'image.

.

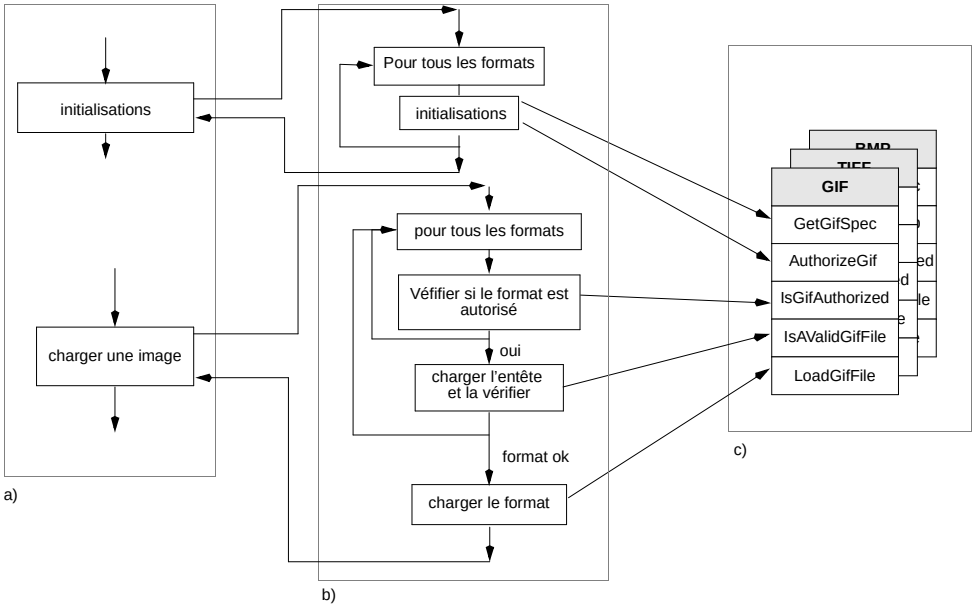

fig 11 : Transparence des formats de fichiers pour l'application principale:
a) Application principale.
b) Couche intermédiare (transparence des formats).
c) Modules "formats images".

On comprend donc qu'il est relativement aisé d'ajouter de nouveaux formats de fichiers. En plus d'écrire le code nécessaire à la manipulation du format en question, il suffit d'ajouter quelques lignes de codes dans la couche qui assure la transparence des formats (c.f. fig 11,b).

4.3 Simplifier le rajout de recherche

Un dernier endroit, et probablement le plus important, où il est indispensable d'écrire le code en vue de mises à jour ultérieures, est constitué par les paramètres de recherche. En effet, il est probable que l'on désirera ajouter d'autres paramètres de recherche par la suite.
On met ici l'accent sur le peu de modifications à apporter au code existant, quitte à compliquer un peu la programmation des paramètres de recherche. Il est en effet plus facile de travailler sur du code compliqué que l'on a écrit soi-même et l'insérer ensuite de manière simple dans du code écrit par une tierce personne, que le contraire.

• Fonctions indispensables

Ici aussi, pour ajouter un module de recherche, on va avoir à coder un ensemble de routines "standards" au comportement bien défini.

CalculXXXImgDatagrm

Où XXX est le nom du paramètre de recherche. Cette fonction, qui reçoit en paramètre une image, a pour mission d'en calculer sa propre interprétation[1] (un histogramme de couleur pour la recherche par couleur) et de rendre le résultat (le *datagramme image*) dans un bloc de données, qui sera manipulé comme une boîte noire par la base de données.

GetXXXUserDatagrm

C'est à cette fonction qu'incombe la tâche de demander poliment à l'utilisateur ce qu'il cherche par l'intermédiaire d'une interface. Notons que, contrairement aux autres fonctions, celle-ci est écrite pour Windows. Elle devra rendre un bloc de données (le *datagramme utilisateur)*, qui lui aussi sera manipulé comme une boite noire par la base de données.

DistanceUDXXX

Calcule la distance entre un datagramme utilisateur et un datagramme image, c'est cette fonction qui est appelée lors de recherche dans la base de données. Rappelons que pour éviter de favoriser des paramètres de recherche par rapport à d'autres, les distances rendues doivent en quelque sorte être normalisées (c.f. 3.2, page 25)

GetXXXErrorMsg

Il est prudent d'éviter que chaque paramètre de recherche affiche de sa propre initiative des messages d'erreurs. Afficher un tel message, implique d'avoir le contrôle de l'interface utilisateur, ce qui n'est le cas que pour la fonction GetXXXUserDatagrm. Il est plus profitable de se limiter à renvoyer un code d'erreur, quitte à ce que l'application récupère par la suite le message correspondant. Et

1. Par exemple: pour la recherche par couleurs, un histogramme 3D (c.f. 5.1, page 39).

c'est justement ce à quoi cette fonction est destinée: retrouver le message correspondant à une erreur qui s'est produite dans son propre module.

XXXInternalInit

Cette fonction n'a qu'une importance pratique. Appelée au début de l'application, elle sert à lancer une éventuelle initialisation du module (calcul du datagramme utilisateur par défaut, par exemple).

GetSpecsOfXXX

C'est dans cette fonction que réside la simplicité de l'ajout d'un paramètre de recherche dans l'application. On aurait pu procéder de la même manière que pour les formats de fichiers (c.f. 4.2), et coder un appel par paramètre de recherche à chaque fois que cela se serait avéré nécessaire. Mais bien que tous ces appels eussent été regroupés dans un seul fichier, les risques d'oubli ou d'erreur auraient été trop grands.

C'est pourquoi cette fonction a été créée: elle a pour but de renvoyer une structure, donnant l'adresse de chacune des fonctions précédentes, ainsi que quelques informations supplémentaires tel que le numéro d'ID qui devra être utilisé pour récupérer les datagrammes dans la base de données (ch 3.2, page 25). Ainsi, pour appeler les autres fonctions du module, l'application n'a qu'à utiliser les adresses obtenues grâce à cette structure. Ce qui évite d'avoir à coder explicitement les appels de fonctions nécessaires à l'utilisation d'un nouveau module. On comprendra donc que pour ajouter un paramètre de recherche, il suffit d'ajouter une seule ligne de code, l'appel de cette fonction GetSpecsOfXXX justement.

Notons que cette méthode, quoique élégante, présente une petite particularité: les déclarations de toutes ces fonctions deviennent alors très strictes. Impossible de modifier le nombre ou la nature des paramètres d'appel de cette fonction, ce qui n'est pas forcément un inconvénient en soi.

Ainsi nous en avons terminé avec le fonctionnement interne de la base de données, il est temps maintenant de discuter de la partie qui justifie la conception de cette application dans un groupe vision: les paramètres de recherches.

5 Recherches

Toute base de données qui se respecte se doit d'offrir des paramètres de recherche pour accéder aux données qu'elle contient. Notre application n'échappe donc pas à la règle, elle propose divers types de recherche qu'il est possible d'étendre (c.f. 4.3):

5.1 Couleurs

La recherche la plus classique est bien sûr la recherche par couleurs: trouver des images qui contiennent une quantité d'une certaine couleur spécifiée par l'utilisateur. On peut aussi avoir à chercher plusieurs couleurs dans la même image.

Tout d'abord considérons la méthode employée pour calculer l'interprétation de l'image en matière de couleurs (i.e. le datagramme image).

- **Histogramme**

  Le but de l'opération est de comptabiliser les couleurs qui se trouvent dans l'image. On a tout d'abord le choix entre plusieurs systèmes de couleurs pour calculer l'interprétation:

  - Munsell (H,V,C), comme le système QBIC qui présente l'avantage de donner de bons résultats [4] mais qui nécessite des conversions assez fastidieuses à l'aide de tables [3],[5].
  - RGB qui lui, a le mérite d'être très simple à employer, c'est généralement selon ce modèle que sont codées les images informatiques.

- HLS communément utilisé en imagerie. C'est ce dernier qui a été choisi parce qu'il est assez "compatible" avec le système de vision humain [11].

Ensuite, le choix de la structure de données pour stocker les couleurs comptabilisées: il n'est bien entendu pas possible de comptabiliser chaque couleur une par une: sur des images 24 bits telles que les manipule notre application, une telle structure prendrait 16Mo (256^3), pour une seule image. On n'a donc guère le choix, il faut créer un histogramme 3D comprenant une dimension pour chacune des composantes H, L, S (c.f. fig 12). Il est bien évident qu'il faut clusteriser cet histogramme, la question est alors de déterminer la taille des clusters: trop petite, l'histogramme prendra beaucoup trop de place, trop grand l'histogramme n'aura plus aucune signification /il correspondra à un simple moyennage de l'image).
Notre choix s'est porté sur un histogramme de 16^3 clusters[1] , contenant chacun une valeur située dans l'interval,e $[0..255]$ indiquent quelle proportion de couleur correspondant à ce cluster, est présente dans l'image. Notons que l'utilisation de puissance de 2 pour les dimensions de l'histogramme n'est pas dûe au hasard, mais répond à des exigences techniques. Il est beaucoup plus facile et rapide de faire des multiplications et des divisions avec des puissances de 2.

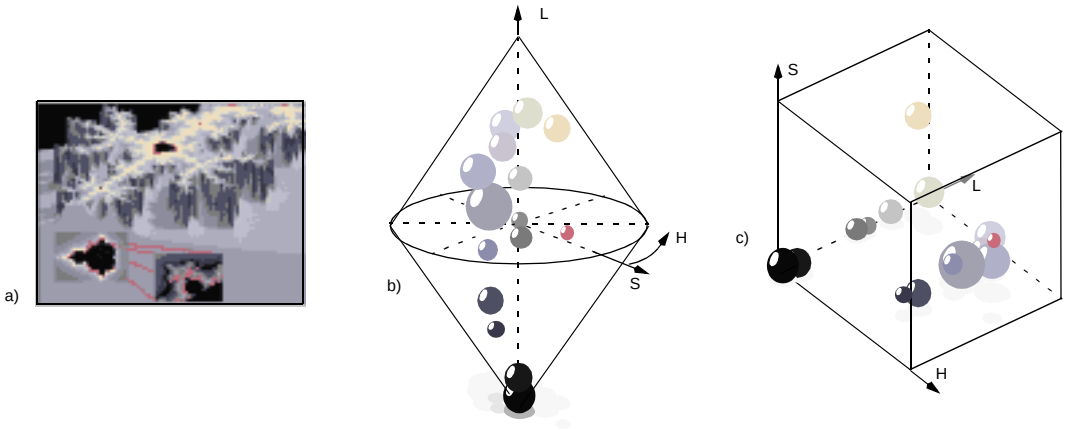

fig 12 : Histogramme H,L,S pour la recherche par couleur:
a) Image à traiter.
b) histogramme HLS correspondant.
c) histogramme HLS mappé sur un cube.[2]

Nous avons donc défini le format de notre datagramme image, il reste donc à définir le datagramme utilisateur, autrement dit celui qui, obtenu par l'intermédiaire de l'interface, va permettre de connaître les désirs de l'utilisateur.

- **Interface**

Le plus naturel consiste à demander à l'utilisateur de fournir une couleur à titre d'exemple. Pour ce faire on fournit une interface (c.f. fig 13)

1. QBIC utilise lui aussi des histogrammes de 16x16x16 clusters
2. Chaque dimension H, L et S est mappée sur un des axes du cube.

dans laquelle il peut spécifier la couleur qu'il désire, soit en la choisissant dans une palette prédéfinie, soit en spécifiant directement les couleurs en coordonnées RGB ou encore HLS.

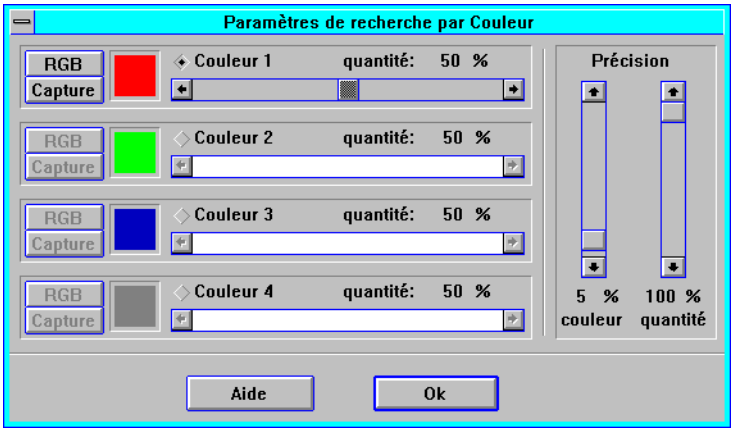

fig 13 : Interface pour le choix des couleurs.

Cependant, à l'usage, on se rend rapidement compte que, à moins d'être très entraîné, il est très difficile de choisir dans une palette une couleur approchant celle d'une image naturelle et plus difficile encore d'en donner les composantes RGB ou HLS. C'est pourquoi il a été ajouté à l'interface, une fonctionnalité suplémentaire: la capture d'une couleur à l'écran, l'utilisateur doit simplement définir un carré sur l'écran dont on calcule la moyenne des composantes RGB des pixels qui la composent. Cela est idéal sur des fonds relativement unis, mais pas du tout sur des fonds variés: une capture effectuée à cheval sur une partie rouge et une partie bleue donnera une couleur violette.

L'utilisateur peut spécifier la quantité de couleurs qu'il désire: grâce à des curseurs (c.f. fig 13). Cependant, après quelques instants de réflexion, on se rend compte que si l'on demande une image qui contienne 50% de la couleur (128,128,0) il y a très peu de chance pour que la recherche aboutisse: il y a 16 millions de couleurs possibles. Il faut donc introduire un facteur d'incertitude. C'est pourquoi il a été décidé d'introduire un facteur de précision sur la couleur et la quantité. Les images résultant d'une recherche sont ensuite triées en fonction de leur fidélité à la quantité désirée spécifiée.

De ce qui précède, on peut déduire la structure du datagramme utilisateur *DtGrmU* renvoyé par l'interface:

$$
\begin{aligned}
DtGrmU = \{ \quad & \{ Actif_1 , (H_1 , L_1 , S_1), Quantité_1\}, \\
& \{ Actif_2 , (H_1 ,L_2 , S_2), Quantité_2 \}, \\
& \{ Actif_3 , (H_3 ,L_3 ,S_3), Quantité_3 \}, \\
& \{ Actif_4 , (H_4 ,L_4 , S_4), Quantité_4 \}, \\
& PrécisionC, \\
& PrécisionQ, \\
\}
\end{aligned}
$$

Recherches

Où $Actif_n$ indique si la couleur n (on se rappelle qu'il est possible de faire une recherche portant sur quatre couleurs simultanément) est active pendant la recherche, (H_n, L_n, S_n) les composantes HLS de la couleur n, $Quantité_n$ est la quantitée (pourcentage) désirée de la couleur n, $PrécisionC$ la précision avec laquelle les couleurs demandées doivent être recherchées et $Précision$Q la précision sur la quantitée.

• Calcul des distances datagramme image/utilisateur

En ce qui concerne les couleurs, nous avons donc décrit les données qui sont extraites d'une image, ainsi que celles qui sont issues de l'interface utilisateur, il ne nous reste alors plus qu'à définir une distance entre ces deux types de données.

Une méthode telle que celle décrite par B.M. Mehtre and Co dans [6], consiste à créer un histogramme 3D à partir du datagramme utilisateur, et à calculer la distance[1] entre ce dernier et l'histogramme contenu dans le datagramme image. Mais cette méthode présente deux inconvénients majeurs. Non seulement le calcul d'une telle distance demande un nombre important d'opérations, mais il va prendre en compte les zones noires créées dans l'histogramme obtenu à partir du datagramme utilisateur. Autrement dit notre application rechercherait en priorité les images qui contiennent *exclusivement* les couleurs demandées, ce qui peut paraître raisonnable lorsque l'on travaille sur des logos[2] mais l'est beaucoup moins lorsqu'il s'agit d'images naturelles.

Il faut donc trouver une autre méthode plus adaptée: c'est pour quoi il a été choisi pour calculer cette distance de faire correspondre à chacune des couleurs désirées (H_n, L_n, S_n), un cube (cluster) I_n dans l'histogramme HLS, centré sur la couleur recherchée et dont la taille est proportionnelle à la précision demandée pour les couleurs (c.f. fig 14).

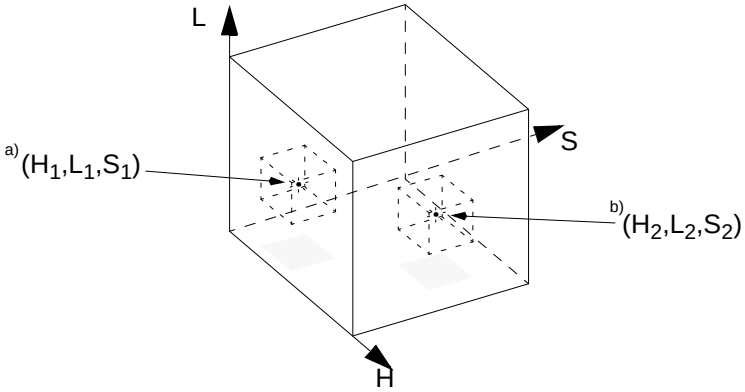

fig 14 : Calcul des distances dans l'histogramme HLS.: clusters:
a) cluster un une couleur 1.
b) cluster pour une couleur 2.

Pour chacune des couleurs désirées (H_n, L_n, S_n), le cluster I_n est balayé de manière à trouver la valeur $V_n(H_n, L_n, S_n)$ de l'histogramme la plus pro-

1. La distance utilisée est la somme des carrés des différences élément par éléments
2. Rappelons que le travail décrit dans [6] vise à retrouver des logos en fonction de leurs couleurs.

che de la quantité désirée pour la couleur n. La différence $D_n = (V_n - quantité_n)$ est filtré par *PrécisionQ*: si $D_n > PrécisionQ$ alors $D_n = 256$. La distance finale est calculée en faisant la somme des carrés de D_n. Finalement, on normalise de manière à ce qu'elle ne dépasse pas *1* . Notons que l'on raisonne comme si HLS était un espace cubique (c.f. fig 12.c), ce qui n'est pas le cas, mais ne nous gène pas.
PrécisionC agit donc sur la taille du cluster balayé dans l'histogramme des couleurs, tandis que *PrécisionC* agit en tant que seuil sur les différences de quantités *Dn*.

Le cluster I_n correspondant à la $n^{ème}$ couleur recherchée, est calculé ainsi:

$$
\begin{aligned}
I_n(DtGrmU) = [\quad & H_n - PrécisionC / 16 ... H_n + PrécisionC / 16, \\
& Ln - PrécisionC / 16 ... Ln + PrécisionC / 16 \\
& S_n - PrécisionC / 16 ... S_n + PrécisionC / 16]
\end{aligned}
$$

Rappelons que les valeurs (H_n, L_n, S_n) sont comprises entre 0 et 255, *PrécisionC* est compris entre 0 et 100 (arbitraire) tandis que la taille de l'histogramme est de 16x16x16 , d'où cette division par 16 (*256/16=16*).

La distance entre le datagramme utilisateur *DtGrmU* et le datagramme image *DtGrmI* est donc calculée par:

$$V_n = \min_{h,\, l,\, s \subset I_n} \left| DtGrmU_{(h,\, l,\, s)} - Quantite_n \right|$$

$$Dn = V_n - Quantite_n \quad si\ V_n - Quantite_n < PrecisionQ$$

$$Dn = 256 \quad sinon$$

$$D(DtGrmU, DtGrml) = \frac{\sqrt{\sum_{n,\, tq\ \ Actif_n = 1} D_n^2}}{NbCoulActives \times 255}$$

Où *NbCoulActives* est le nombre de couleurs actives, autrement dit, le nombre de couleurs que l'on recherche.

Il ne reste donc plus qu'à calculer cette distance lors des recherches et ne conserver que les images (dans notre cas, les 32 meilleures) qui rendent la valeur de distance la plus faible.

• Résultats

Les résultats de cette méthode sont dans l'ensemble assez satisfaisants: on trouvera quelques illustrations de résultats de recherches par couleur à la fig 15.

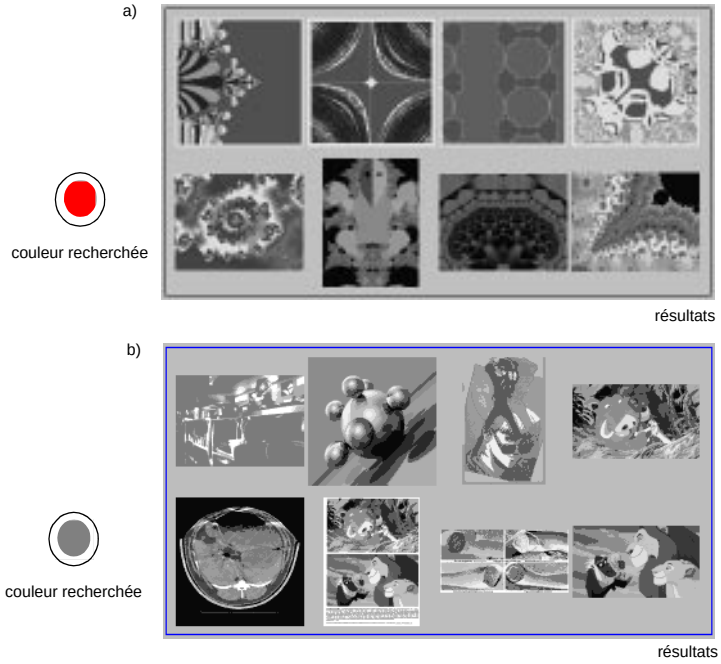

fig 15 : Recherche par couleurs: résultats:
a) recherche de rouge, précisionC 5%, quantité 50%, précisionQ 50%.
b) recherche de gris, précisionC 5%, quantité 90%, précisionQ 50%.
.

Cependant, quelques problèmes peuvent surgir, notre méthode fait preuve de quelques petites faiblesses:

Images tramées:
Cet algorithme ne marche pas lorsqu'il s'agit de rechercher des images tramées avec des couleurs sensiblement différentes. Par exemple, du type de celles qui ont été décrites au chapitre 2.3. Si l'on considère un exemple typique: une image constituée exclusivement de pixels rouges et bleus disposés en damier apparaîtra comme violette à l'observateur, alors qu'une recherche sur le violet ne la trouvera jamais. Une façon, peu élégante il est vrai, de résoudre le problème serait de passer un filtre moyenneur sur l'image avant d'en calculer son datagramme image pour la couleur. Mais signalons tout de même que ce type de situation est très rare en particulier lorsque l'on a affaire à des images 24 bits[1].

Nous avons maintenant fait le tour de la question en ce qui concerne la recherche par couleur: il est temps de s'occuper de la recherche par texture.

1. Cette technique de tramage, avec des couleurs très différentes dans la matrice de génération de la trame, est généralement utilisée pour générer des pseudo-couleurs avec une palette très réduite (16 couleurs).

5.2 Textures

La recherche par textures, plus complexe que la recherche par couleurs, consiste à retrouver des images contenant une texture définie par l'utilisateur.

Une texture peut être définie comme la structure détaillée d'une image, trop fine pour être analysée entièrement, cependant assez grossière pour induire une fluctuation notable des niveaux de gris de pixels voisins [10].

Insistons sur le terme "niveaux de gris". En effet l'analyse de textures se fait traditionnellement sur des niveaux de gris. On serait très tenté de réfléchir à une méthode qui permettrait de retrouver des textures colorées, cependant, un tel type de recherche risquerait fort de faire double emploi avec la recherche par couleurs.

Tout d'abord, il nous faut donc déterminer la méthode à employer, pour créer un datagramme image qui réponde à ces deux critères:

- Compacte.
- Rapide à calculer.

• Calcul des caractéristiques

Il existe de nombreuses méthodes pour calculer les caractéristiques d'une texture, et bien-sûr toutes ont leurs avantages et leurs inconvénients.

La transformée de Gabor

Les fonctions de Gabor [7][10], généralisation des fonction de base de transformées de Fourier, permettent de retrouver les caractéristiques d'une image dans le plan fréquentiel. Chaque fonction est en général utilisée comme un filtre pour une région spécifique de ce plan fréquentiel, ce qui rend en théorie l'analyse des résultats assez facile.
Une telle méthode parait séduisante au premier abord. Cependant, elle présente un grave inconvénient: elle demande des calculs très lourds sur des nombres en virgule flottante.

La segmentation

Une autre approche, consiste à segmenter la texture comme on le ferait d'une simple image, pour ensuite en faire une analyse structurelle. Cette méthode très lourde, n'est valable que pour des textures fortement ordrées, c'est-à-dire des textures qui présentent une structure claire et régulière (c.f. fig 16).

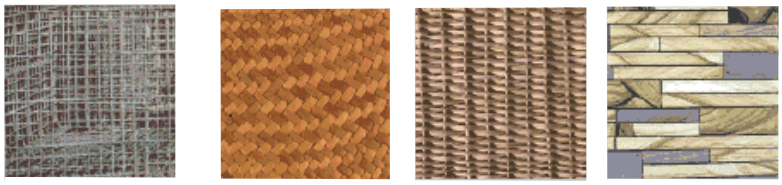

fig 16 : Quelques textures fortement ordrées

L'analyse statistique

Les méthodes statistiques permettent une analyse simple et relativement fiable d'une texture [9]. On distingue les méthodes de premier ordre, qui portent sur des calculs à partir de l'histogramme des niveaux de gris, et les méthodes de second ordre, qui elles, travaillent sur la densité moyenne des contours, la longueur typique des plages de même intensité, les histogrammes généralisés à des primitives ou encore les matrices de co-occurrence généralisées [10].

• Les méthodes de premier ordre

Les méthodes de premier ordre se calculent à partir de l'histogramme des niveaux de gris de la texture, qui est calculé comme

$$h(i) = \frac{\phi(i)}{A} \qquad (1)$$

Où $\phi(i)_{i = 0.255}$ est le nombre de pixels dans l'image ayant l'intensité i et A le nombre total de pixels dans l'image. De ces valeurs $h(i)$ on peut tirer les valeurs suivantes [9]:

La moyenne (Mean):

$$\mu = \sum_{i=0}^{255} (i+1)h(i) \qquad (2)$$

La déviation standard (Standard Déviation):

$$\sum_{i=0}^{255} (i+1-\mu)^2 h(i) \qquad (3)$$

Le troisième moment[1] (Third Moment):

$$\sum_{i=0}^{255} (i+1-\mu)^3 h(i) \qquad (4)$$

1. Le troisième moment permet de mesurer la symétrie de l'histogramme: quand l'histogramme est décalé sur la droite ou la gauche, la valeur de ce paramètre est négative ou positive, si au contraire, l'histogramme est symétrique, cette valeur est nulle.

L'entropie[1] (Entropy):

$$-\sum_{i=0}^{255} h(i)\log h(i) \qquad (5)$$

• Les méthodes de second ordre:

Il a été choisi pour implémenter les méthodes de deuxième ordre d'utiliser les matrices de co-occurrence. Cette matrice, notée P_δ, permet de comptabiliser les transitions d'un niveau de gris n_1= *(0...255)* à un niveau de gris n_2 = *(0..255)* entre des pixels séparés par un vecteur $\delta = (r, \theta)$. En théorie, il faut calculer toutes les matrices de co-occurrence possibles dans l'image, ce qui est inconcevable. Haralick, Shanmugam et Dinstein (1973) qui furent les premiers à utiliser les matrices de co-occurrence afin d'analyser des photographies aériennes se contentèrent de calculer les matrices pour $\theta = 0°, 45°, 90° et 135°$ et *r=1*. Toujours pour des raisons d'efficacité, les matrices ont été réduites à une taille de 64x64 éléments au lieu de 255x255.

Exemple de calcul de P_δ avec $\delta = (1, 41;45°)$: et *IM* une image (en niveaux de gris) de dimentions *LxH:*

$P_\delta(i, j) = 0$, $\forall i \in 0\ldots63$, $\forall j \in 0\ldots63$

Pour chaque i tel que $i \in [1\ldots L-1]$ *boucler*

Pour chaque j tel que $j \in [1\ldots H-1]$ *boucler*

incrémenter $P_\delta(IM((i, j), IM(i+1, j+1)))$

fin boucler

fin boucler

Pour chacune de ces matrices, nous allons donc calculer [9]:

L'énergie[2] (Angular Second Moment):

$$\sum_{i=0}^{63}\sum_{j=0}^{63} P_\delta(i+1, j+1)^2 \qquad (6)$$

1. L'entropie mesure l'uniformité de l'histogramme: plus l'histogramme est uniforme plus cette valeur approche de la valeur maximale, si par-contre il contient un cluster dense, cette valeur s'approche de 0.

Le contraste[1] (Constrast):

$$\sum_{k=0}^{63} k^2 \sum_{\substack{i=0 \\ |i-j| = k}}^{63} \sum_{j=0}^{63} P_\delta(i+1, j+1) \tag{7}$$

La Corrélation[2] (Correlation)

$$\frac{\sum_{i=0}^{63} \sum_{j=0}^{63} (i+1)(j+1) P_\delta(i+1, j+1) - \mu_x \mu_y}{\sigma_x \sigma_y} \tag{8}$$

où

$$\mu_x = \sum_{i=0}^{63} i \sum_{j=0}^{63} P_\delta(i+1, j+1) \tag{9}$$

$$\mu_y = \sum_{j=0}^{63} j \sum_{i=0}^{63} P_\delta(i+1, j+1) \tag{10}$$

$$\sigma_x = \sum_{i=0}^{63} (i+1-\mu_x)^2 \sum_{j=0}^{63} P_\delta(i+1, j+1) \tag{11}$$

$$\sigma_y = \sum_{j=0}^{63} (j+1-\mu_x)^2 \sum_{i=0}^{63} P_\delta(i+1, j+1) \tag{12}$$

A partir des matrices de co-occurrence il est aussi possible de calculer les "Difference Statistics" [traduction?] qui sont la distribution des probabilités $P_\delta(k)$ *(k=0,...,63)* pour que la différence de niveaux de gris soit k entre les points séparés par δ dans l'image. Les "Difference Statistics" sont calculées à partir des matrices de co-occurrence de la manière sui-

2. l'énergie mesure l'homogénéité de la texture: moins la matrice aura des entrées de grande magnitude plus l'énergie sera grande.

1. Le contraste permet de mesurer la valeur des variations locales dans l'image.

2. Une corrélation importante dans une direction θ signifiera que l'image contient une quantité importante de structures linéaires orientées dans cette direction.

vante[9]:

$$P_\delta(k) = \sum_{\substack{i=0 \\ |i-j| = k}}^{63} \sum_{j=0}^{63} P_\delta(i+1, j+1) \qquad (13)$$

Il est alors possible de calculer les propriétés suivantes à partir de $P_\delta(k)$.

Le second moment angulaire (Second angular moment):

$$\sum_{k=0}^{63} P_\delta(k)^2 \qquad (14)$$

Le contraste (Constrast):

$$\sum_{k=0}^{63} k^2 P_\delta(k) \qquad (15)$$

L'entropie (Entropy):

$$-\sum_{k=0}^{63} P_\delta(k) \log P_\delta(k) \qquad (16)$$

La moyenne: (Mean)

$$\sum_{k=0}^{63} k P_\delta(k) \qquad (17)$$

La figure numérotée fig 17 illustre quelques exemples de textures dont

l'histogramme et les quatre matrices de co-occurrence ont été dessinées, On remarquera que plus une texture est uniforme (c.f. fig 17b), plus les élements des matrices de co-occurence forment une tache ronde. Alors qu'au contraitre si la texture semble avoir une direction de prédilection (c.f. fig 17c), les éléments de la matrice correspondante forment une tache allongée.

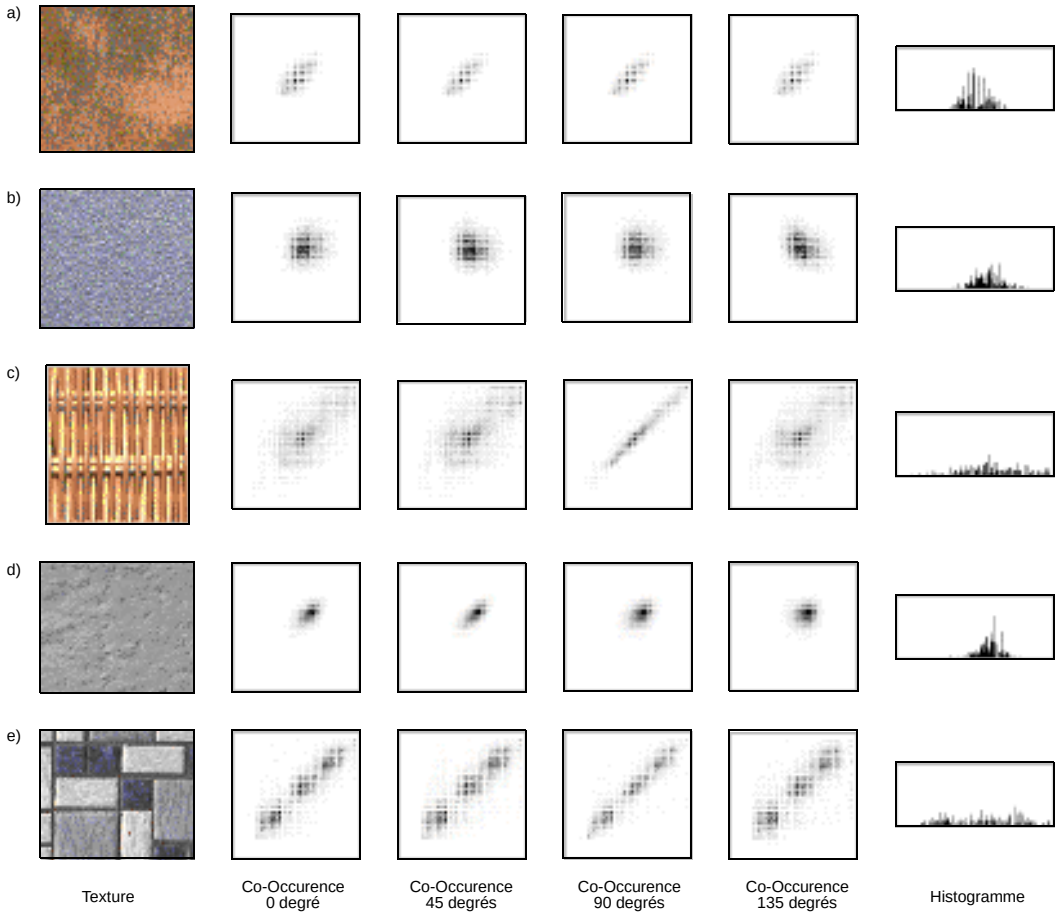

fig 17 : textures, matrices de co-occurence et histogrammes.

On comprend donc que l'on dispose d'un vecteur de $4+4x7=32$ nombres en virgule flottante pour décrire une texture (c.f. formules (2,3, 4, 5) et (6, 7, 8 , 14, 15, 16, 17)).

Cependant un problème peut apparaître. Les éléments du vecteur de description ne sont pas du même ordre de grandeur, ce qui va poser des problèmes. Lors du calcul des distances, certaines dimensions risquent d'être systématiquement privilégiées. C'est pourquoi il a été décidé de les diviser systématiquement par la moyenne de chaque élément de ce vecteur qui a été calculée sur un ensemble de plus de 8000 textures[1]. Ainsi tous les éléments du vecteur ont été ramenés à un ordre de grandeur semblable, à savoir 1.

1. Elles provenaient de 500 images découpées en carrés de 100x100.

Le calcul d'un vecteur pour une image entière pourrait suffire, si chaque image qui devait être dans la base était une texture pure. Malheureusement ce n'est vraisemblement pas le cas (la plupart des images ne sont pas des "photos" de textures). Aussi ne peut-on donc pas se permettre une telle simplification, les mesures effectuées sont immanquablement perturbées par les éléments non texturés de l'image.

La parade consiste à partitionner l'image en petits carrés, et calculer un vecteur pour chacun de ces carrés. Mais la taille de ces carrés doit être judicieusement choisie. En effet, trop petits, les vecteurs nécessaires à la description d'une image seraient trop nombreux; trop grands, le problème décrit précédemment réapparaitrait: on aurait toujours des carrés pouvant contenir plusieurs textures. Après expérimentations, la taille de ces carrés a été fixée à 100 pixels de côté.

Mais un simple calcul nous apprend qu'une image de taille moyenne 512x512 pixels peut fournir 25 de ces vecteurs, qui eux-mêmes ont une taille de 32x8 = 256 octets, ce qui donne environ 6,5 Ko uniquement pour décrire la texture d'une image. De plus, nombre de ces vecteurs vont décrire des morceaux d'images, qui ne seront pas toujours des textures mais pourront être formes significatives telles des visages, des morceaux de roue de voiture etc. Ce qui conduit à un gaspillage d'espace disque.
Il se trouve que le paramètre corrélation, calculé à partir des matrices de co-occurrence a curieusement tendance à prendre des valeurs fantaisistes[1] lorsque l'on essaye de le calculer à partir d'une image qui n'est pas une texture. Après quelques expérimentations, et notamment la visualisation de 3000 carrés 100x100 qui n'étaient pas considérés comme des textures par l'application, il a été décidé de tirer parti de cette particularité.
A l'usage, il apparaît que sur une base de données qui contient des images qui ne sont pas des textures "pures" (ce qui ne signifie par forcément qu'elles n'en contiennent pas), on gagne environ 40% en place et le temps de recherche se trouve diminué dans des proportions identiques.

• Interface

De même que pour la recherche par couleurs, une interface est nécessaire pour que l'utilisateur puisse spécifier les textures qu'il désire retrouver dans la base de données.

Générer une texture artificielle

Pour rechercher les images en fonction des textures qu'elles contiennent, le seul moyen qui vienne à l'esprit est de demander à l'utilisateur de fournir une texture à titre d'exemple puis de rechercher toutes les images qui comportent cette texture. C'est pourquoi notre application possède une interface qui permet de générer des textures à partir d'un profil (c.f. fig 18), que l'utilisateur peut modifier à son gré: en ajoutant ou supprimant des points, en déplaçant des parties du profil, etc.

1. Pour une image non texturée le paramètre corrélation prend des valeurs supérieures à 10^{12} alors qu'en général, il est voisin de 1.

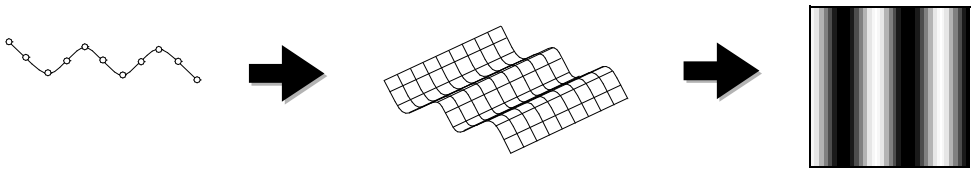

fig 18 : Génération d'une texture à partir d'un profil.

Notons qu'il peut ensuite être intéressant de faire pivoter la texture ou encore de faire varier la fréquence ou la phase selon laquelle le profil est employé (c.f. fig 19).

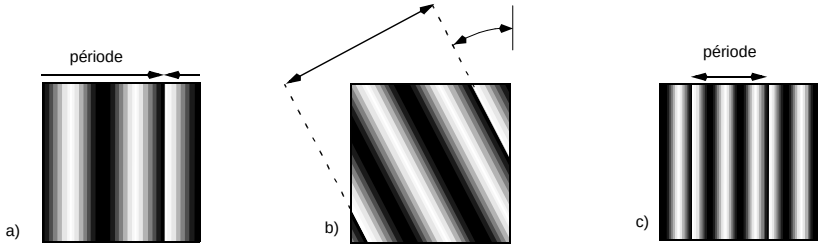

fig 19 : Modification de la texture générée:
a) modification de la phase.
b) modification de l'angle.
c) modification de la fréquence.

Bien qu'intéressants ces résultats sont un peu insuffisants: les images obtenues ne ressemblent guère à une texture, mais en combinant différentes textures générées de cette manière, il est possible de créer des images plus évoluées. Parmi toutes les combinaisons possibles, deux semblent particulièrement attrayantes:

• Combinaison multiplicative:

Il s'agit de multiplier chaque pixel d'une des textures avec la valeur de son vis à vis dans l'autre texture, puis de normaliser le résultat. La méthode est similaire à un "ET" logique. si $T1$ est une texture de dimension $H1xL1$ et $T2$ une texture de dimensions $H2xL2$, la texture résultante $T3$ aura pour dimensions $H3 = min(H1,H2)$ et $L3 = min(L1,L2)$, et les valeurs de ses pixels seront:

$$(T3_{i,j})_{\substack{j = 1 \ldots min(H1,H2) \\ i = 1 \ldots min(L1,L2)}} = \frac{T1_{i,j} \times T2_{i,j}}{65535}$$

Ce type de combinaison est particulièrement utile pour créer des structures rondes ou ovales (c.f. fig 20).

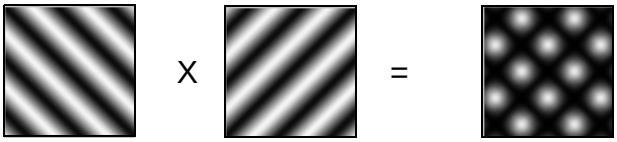

fig 20 : Combinaison multiplicative de textures.

• Combinaison additive:

Il s'agit de comparer chaque pixel d'une des textures avec la valeur de son vis à vis dans l'autre texture, puis de conserver la plus élevée, La méthode est similaire à un "OU" logique. Si $T1$ est une texture de dimensions $H1xL1$ et $T2$ une texture de dimensions $H2xL2$, la texture résultante $T3$ aura pour dimensions $H3 = min(H1,H2)$ et $L3 = min\ (L1,L2)$. Les valeurs de ses pixels sont obtenus grâce à la formule suivante:

$$(T3_{i,j})_{\substack{j = 1 \ldots min(H1,\, H2) \\ i = 1 \ldots min(L1,\, L2)}} = max(T1_{i,j}, T2_{i,j})$$

Cette combinaison permet de créer des quadrillages (c.f. fig 21) dans la texture, ce qu'interdisait l'opération précédente.

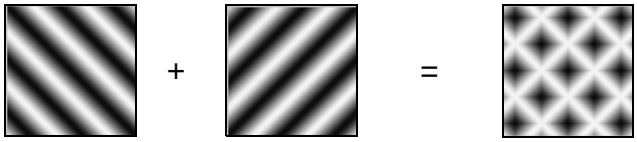

fig 21 : Combinaison additive de textures.

De plus il peut être intéressant de modifier la texture finale grâce à diverses opérations telles que:

• La translation (cyclique)

Bien qu'en principe cela ne change rien aux mesures statistiques qui seront faites sur la texture, la translation peut s'avérer très utile pour faire centrer l'image avant de combiner deux textures.

• Modifier la luminosité

Il peut être intéressant de modifier la luminosité d'une texture. Soit une texture T de dimensions HxL, et un coefficient C, plus petit que 1 pour diminuer la luminosité, plus grand pour l'augmenter. On calcule $T2$ de mêmes dimensions que T:

$$(T2_{i,j})_{\substack{j = 1 \ldots H \\ i = 1 \ldots L}} = C \times T_{i,j} \qquad 0 \le T_{i,j} \le 255$$

• Modifier le contraste

Il peut s'avérer utile de modifier le contraste d'une texture. Toujours avec une texture T de dimension HxL: on calcule le niveau de

gris moyen μ. Puis on écarte (ou rapproche) les valeurs des pixels de T de cette moyenne. Soit C un coefficient, plus petit que 1 pour diminuer le contraste, plus grand pour l'augmenter, on calcule $T2$ de mêmes dimensions que T.

$$\mu = \frac{\sum_{i=1}^{H}\left(\sum_{j=1}^{L} T_{i,j}\right)}{HxL}$$

$$(T2_{i,j})_{\substack{j=1\ldots H\\ i=1\ldots L}} = C \times (T_{i,j} - \mu) + \mu \qquad 0 \le T_{i,j} \le 255$$

• Atténuer les contours

Un filtre passe bas est une option qui peut s'avérer des plus utiles. Le filtre implémenté est un filtre moyenneur 3x3. Toujours avec T de dimensions HxL, on calcule $T2$ de dimensions $H2=H-2$, $L2= L-2$:

$$(T2_{i,j})_{\substack{j=1\ldots H-2\\ i=1\ldots L-2}} = \frac{T_{i,j} + T_{i+1,j} + \ldots + T_{i+1,j+2} + T_{i+2,j+2}}{9}$$

• Extraire les contours

L'utilisateur peut avoir besoin d'extraire les contours d'une texture, ce qui est implémenté à l'aide du module des gradients horizontaux et verticaux [10]. Toujours avec T de dimensions HxL, on calcule $T2$ de dimensions $H2=H-1$, $L2= L-1$:

$$(T2_{i,j})_{\substack{j=1\ldots H-1\\ i=1\ldots L-1}} = \frac{\sqrt{(T_{i,j} - T_{i+1,j})^2 + (T_{i,j} - T_{i,j+1})^2}}{\sqrt{2}}$$

• Inverser la texture

On peut, dans certains cas vouloir calculer l'inverse d'une texture, toujours à partir de T on calcule $T2$, de mêmes dimensions:

$$(T2_{i,j})_{\substack{j=1\ldots H-1\\ i=1\ldots L-1}} = 255 - T_{i,j}$$

• Rotation:

Enfin, l'utilisateur peut avoir besoin de faire pivoter la texture finale. Comme cette texture est un bitmap, seules les rotations par

pas de 90 degrés sont réalisables; des rotations par pas plus faible provoqueraient des “trous” dans les coins de l’image (c.f. fig 22c).

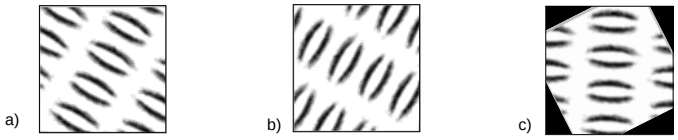

fig 22 : Rotation d’une texture:
a) image originale.
b) rotation de 90 degrés.
c) rotation arbitraire.

Avec une texture de départ T de dimensions HxL, on calcule $T2$ de dimensions $H2=L$, $L2=H$:

$$(T2_{i,j})_{\substack{j=1\ldots L \\ i=1\ldots H}} = T_{j,i}$$

En dépit du fait que cette façon de générer des textures soit basée sur un concept extrêmement simple, elle permet à l’aide de diverses combinaisons de créer des textures très variées (c.f. fig 23).

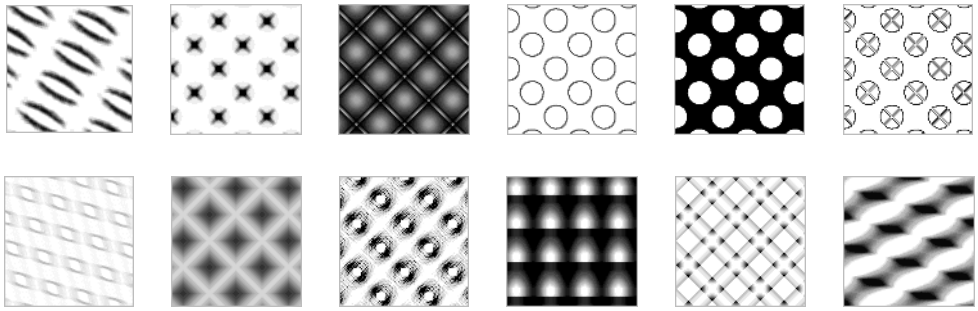

fig 23 : Textures générées artificiellement..

Importer une texture:

Malgré tout, cette méthode pour générer des textures artificielles ne permet pas de créer n’importe quelle texture seule des textures “périodiques” peuvent être générées. Un marbrage par exemple ne peut pas être généré de cette manière. C’est pourquoi, il peut être intéressant de récupérer une texture ailleurs, typiquement en l’important depuis une autre application par l’intermédiaire du clipboard (copier - coller), ou encore, en la capturant sur l’écran.

Notons simplement que l’image capturée sera vraissemblablement en couleur, il faudra donc la convertir en niveaux de gris, R, G, B étant les plans rouge, vert et bleu de l’image originale (de dimensions LxH). On cal-

cule la texture T en niveaux de gris de la manière suivante:

$$(T_{i,j})_{\substack{j=1\ldots min(H,100)\\ i=1\ldots min(L,100)}} = 0.59R_{i,j} + 0.30G_{i,j} + 0.11B_{i,j}$$

Mémorisation des textures

Il peut être intéressant de pouvoir sauvegarder les textures créées, c'est pourquoi l'interface offre une liste où l'utilisateur peut sauver ses textures et bien évidemment les récupérer. Notons que cette récupération peut s'effectuer non seulement comme une simple copie, mais aussi comme une combinaison avec la texture courante.

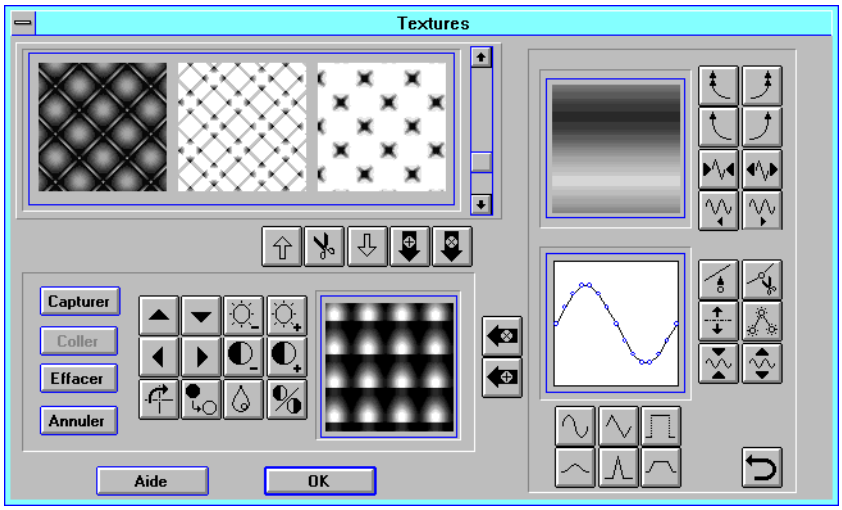

fig 24 : Interface pour la recherche par textures.

Nous avons donc passé en revue tous les éléments qui permettent de créer une interface pour les textures (c.f. fig 24). Le dernier point consiste à déterminer quels type de données cette interface renvoit à l'application principale, une fois que l'utilisateur aura cliqué "Ok". La réponse est simple: il s'agira d'un datagramme utilisateur (vecteur de 32 réels) calculé à partir de la texture générée de la manière décrite au début de ce paragraphe.

• Calcul des distances

Rappelons que pour chaque image, nous disposons d'un nombre n de vecteurs descripteurs V_i , $i=1..n$, chaque vecteur V_i étant constitué de 32 réels $V_{i,j}$ $j=1..32$. De plus nous disposons du vecteur datagramme utilisateur $DtGrnU$ (constitué de 32 réels $DtGrnU_i$ $i=1..32$). La valeur renvoyée par la mesure de distance sera la plus petite distance entre le vecteur datagramme utilisateur $DtGrnU$ et les vecteurs V_i , $i=1..n$, qui forment le datagramme image $DtGrnI$:

$$D(DtGrmU, DtGrml) = \min_{i=1\ldots n}\left(\sqrt{\sum_{j=1}^{32}(V_{i,j} - DtGrmU_j)^2}\right)$$

- **Résultats**

Bien que très simple cette méthode donne des résultats visuellement acceptables (c.f. fig 25) tant que l'on reste dans les limites du raisonnable, autrement dit lorsque l'on travaille avec des textures qui apparaissent comme des images texturée.

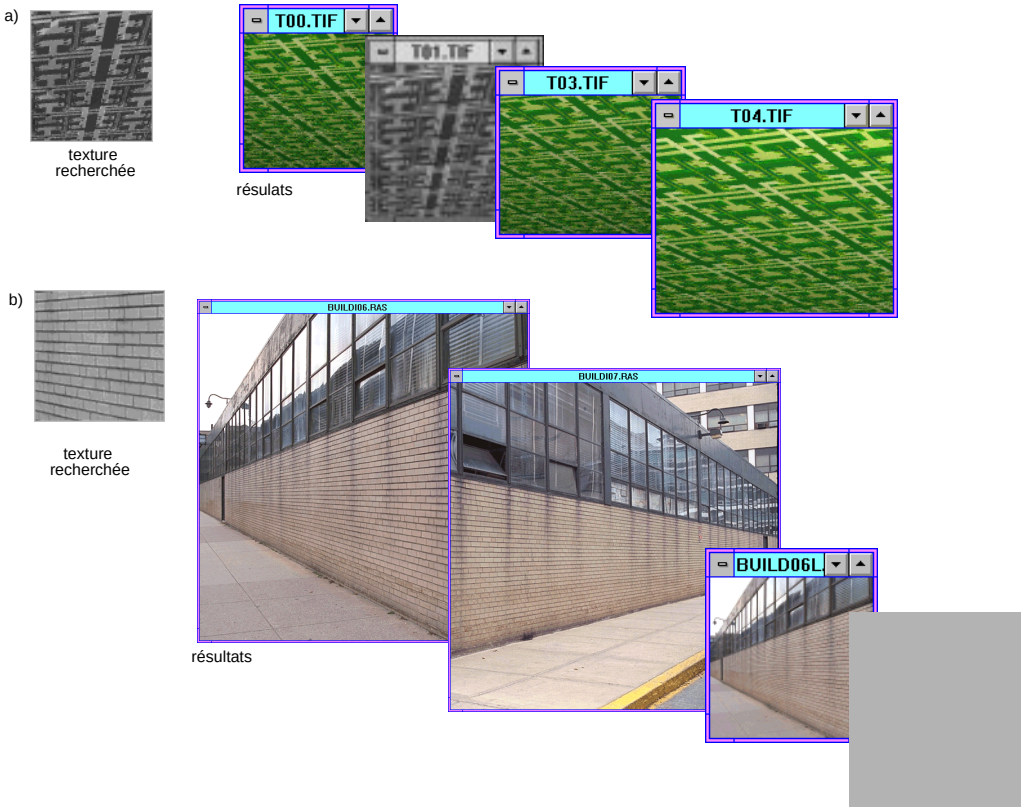

fig 25 : Quelques recherches par textures.

Notons que la manière dont est calculée la distance entre deux vecteurs descripteurs de texture entraîne une certaine invariance à la rotation, en particulier celle de 90 degrés. Par ailleurs, on peut remarquer que dans certains cas bien précis[1] (c.f. fig 25), la méthode employée est insensible à l'homothétie ("scaling").

Cependant, des problèmes commencent à apparaître lorsque les motifs de la texture recherchée commencent à devenir trop gros. Il n'y a alors

1. Lorsque le changement d'echelle n'affecte pas la qualité de l'image.

plus assez de ces motifs dans un carré de 100x100 pixels pour effectuer des mesures statistiques fiables.

Cependant, un problème plus crucial fait sont apparition au fur et à mesure que l'on rajoute des images dans la base de données. Etant donné que la méthode employée est basée sur des statistiques de premier et de second ordre, rien ne garanti que toutes les images de la base, possédant un vecteur de description identique à celui du datagramme image, contient à coup sûr une texture semblable à celle recherchée. En fait, ces exceptions existent [10] et agrandir la base de données semble augmenter considérablement ce risque de "mauvaises rencontres" (c.f. fig 26). Il arrive même qu'il soit impossible de retrouver certaines textures à partir d'un fragment de les mêmes textures. Malheureusement, lorsque ce problème est apparu, pendant les tests avec de grandes bases de données (>1500 images), il était beaucoup trop tard pour y remédier.

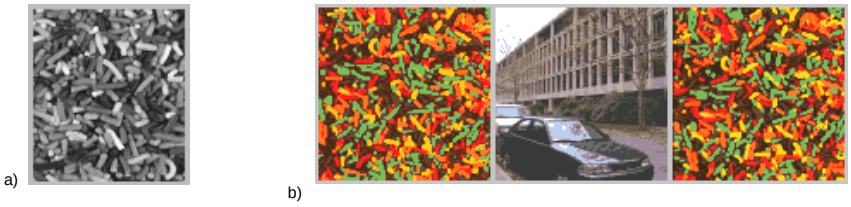

fig 26 : recherche par textures: problème.
a) texture recherchée.
b) résultats: la deuxième image est visiblement fausse.

5.3 Noms

Bien que n'étant pas tellement du ressort de l'analyse d'image, ce type de recherche peut s'avérer très utile: en effet, le nom de fichier est souvent significatif en regard du contenu de l'image qu'il contient.

• Les datagrammes Noms

Les datagrammes noms ne sont pas compliqués: le nom de fichier de chaque image, sans chemin d'accès et sans extension, est stocké dans la base.

• Recherches par noms et distances

Il est possible d'effectuer plusieurs types de "sous-recherche par nom". Supposons que l'utilisateur donne dans l'interface une chaîne de caractères ST de n caractères et que les datagrammes image de la base soient des chaînes caractères $ST2$ de longueur $lg(ST2)$, on peut effectuer une recherche:

Exacte

Autrement dit on recherche une correspondance exacte entre ST et $ST2$, on renverra donc une distance de 0 si les deux chaînes de caractères ne sont pas exactement les mêmes, et 1 dans le cas contraire.

Par début de chaîne

Ici on recherche les noms de fichiers dont les *n* premiers caractères constituent une chaîne de caractères qui soit égale à *ST*. S'il y a correspondance exacte entre les n premiers caractères de *ST2* et *ST* on renverra une distance de *1*, dans le cas contraire on renverra *0*.

Par sous-chaîne

Il s'agit là de rechercher les noms de fichiers qui contiennent la chaîne de caractères *ST:* s'il existe une sous-chaîne de *n* caractères dans le datagramme utilisateur *ST2* qui corresponde exactement à *ST1*, on renvoie une distance d'autant plus proche de *0* que cette sous-chaîne est proche du début de *ST2*. S'il n'existe pas de telle sous-chaîne de caractères dans *ST2*, on renvoie une distance égale à *1*. Ce qui peut s'exprimer la manière suivante:

si $\exists j \in [1 \ldots lg(ST2) - n]$ tq $\forall i \in [1 \ldots n], ST2[i + j - 1] = ST[i]$

$$D = 1 - \left(\frac{9}{10}\right)^{j-1}$$

sinon

$$D = 0$$

Par approximation

DOS, système d'exploitation sur lequel repose Windows, présente une particularité assez déplaisante: les noms de fichiers que l'on emploit sont limités à 8 caractères, on est donc souvent amené à utiliser des noms de fichiers plus courts qu'on ne le voudrait. Une des méthodes pour raccourcir un nom de fichier consiste à en supprimer les voyelles. En ce cas, les recherches décrites plus haut vont s'avérer inopérantes, c'est pourquoi il peut être intéressant d'offrir une recherche qui trouve des chaînes approximatives.

On renvoie donc une distance d'autant plus proche de *0* que *ST2* contient les mêmes lettres que *ST1* et si possible dans le bon ordre. Cette fois-ci la distance ne peut être calculée directement; l'emploi d'un algorithme s'avère nécessaire.

Cet algorithme est basé sur les suites de lettres identiques dans *ST1* et *ST2*. On prend une valeur *res* égale à 1 par défaut, et on parcoure les deux chaînes de caractères, à chaque fois que deux lettres coïncident, *res* est multiplié par une valeur supérieure à 1, qui augmente; dans le cas contraire, *res* est multipliée par une valeur inférieure à 1 qui décroît.

si $ST \subset ST2$ *ou* $ST2 \subset ST$ *alors distance = 0*
sinon
    res = 1.0
    start = 0
    Pour chaque i tel que $i \in [1 \ldots lg(ST)]$ *boucler*
        pénalité = 0.9
        Pour chaque j tel que $j \in [start \ldots lg(ST2)]$ *et* $sti \neq ST2j$ *boucler*
            *res=res*pénalité*
            *pénalité = pénalité*0.9*
        fin boucler
        si j<lg(ST2)
            start = j+1
            *res = res * 1.1*
        sinon
            sortir boucle
        fin si
    fin boucle
    si $lg(ST) \neq lg(ST2)$
        pénalité = 0.9
        Pour chaque i tel que $i \in [1 \ldots |lg(ST) - lg(ST2)|]$ *boucler*
            *res= res*pénalité*
            *pénalité = pénalité * 0.94*
        fin boucle
    fin si
    si res >1.0
        res = 1.0
    fin si
    distance = 1.0 - res.
fin si.

- **Résultats**

En ce qui concerne les trois premiers types de recherche par nom, à savoir, recherche exacte, par début de chaîne, ou par sous-chaîne, les résultats se passent de commentaires.
En ce qui concerne la recherche approximative, on obtient des résultats assez satisfaisants, par exemple:

Recherché: "Diamand"
Trouvé: "diamond", "dangling".

Recherché: "Ballon"
Trouvé: "balls3", "balls2", "ball", "balls", "ballbox1".

Recherché: "Tour-Eiffel"
Trouvé: "eiffel", "eiffel1", "office14", "office13".

6 Performances

A ses débuts, SPIP était développé sur un 486-33, ce qui obligeait à quelques acrobaties au niveau de la programmation, qui par la suite se sont révelées inutiles sur un Pentium 90. Néanmoins, ces optimisations sont restées en place.

6.1 Emplois de buffers

Une des méthodes classiques pour l'optimisation d'une application effectuant beaucoup d'accès disque consiste à bufferiser ces accès disques. Lorsque des données doivent être obtenues depuis les disques, elles sont chargées en mémoire et conservées dans cette dernière le plus longtemps possible au cas où elles pourraient être à nouveau sollicitées. Il existe de nombreuses techniques de buffering, celle employé dans SPIP reste assez simple.

Une file d'attente (buffer) est mise en place. Lorsque des données sur disques doivent être accédées, le buffer est vérifié de manière à s'assurer que les données requises s'y trouvent.

- Si c'est pas le cas, on déplace l'élément concerné en début de file (c.f. fig 27a),
- Dans le cas contraire, si le buffer est plein, le dernier élément de la liste est supprimé, puis l'élément désiré est chargé en début de liste (c.f. fig 27b).

Il ne reste plus qu' à fournir une copie de l'élément concerné à la routine qui en a fait la demande.

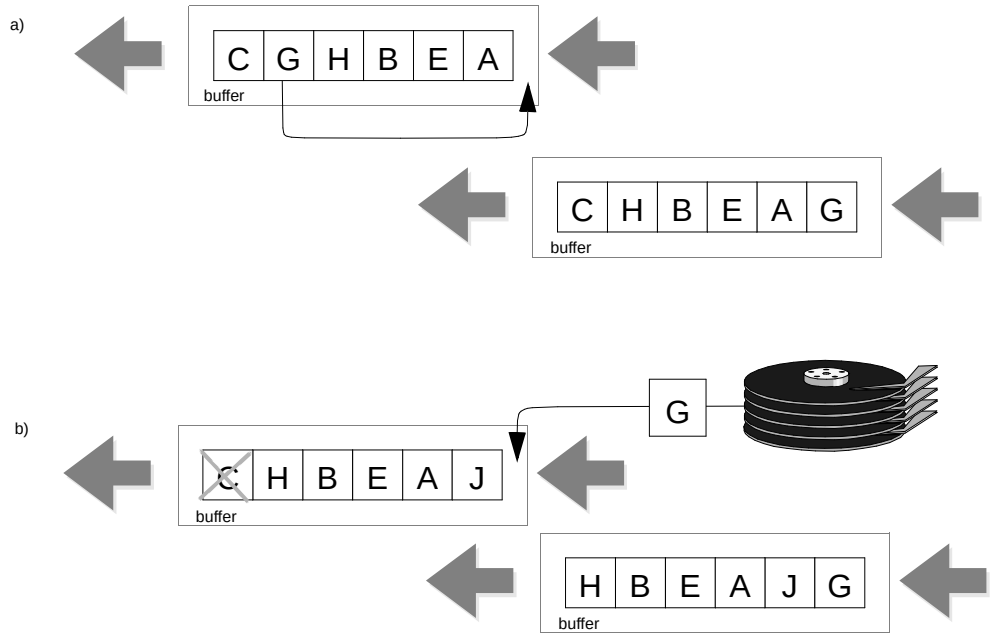

fig 27 : Buffering: charger l'élément "G":
a) élement déja présent dans le buffer.
b) élément absent du buffer.

Bien entendu, lorsque des modifications doivent intervenir sur les disques, les éléments du buffer doivent être mis à jours. Pour forcer cela, le moyen le plus simple consiste à vider les buffers concernés. Cela peut paraître assez abrupt. Mais les accès en écriture sont ici beaucoup plus rares que les accès en lecture.

- **Accès aux descripteurs et aux datagrammes**

  L'accès aux descripteurs d'images dans la base fait partie des éléments bufferisés de la base. En effet, lors des recherches, les descripteurs peuvent être réclamés plusieurs fois de suite puisque ce sont eux qui contiennent la table des datagrammes (c.f. 3.2, page 25). De plus lorsqu'un datagramme correspondant à un descripteur est chargé en mémoire, il est rattaché à ce descripteur et reste en mémoire jusqu'à ce qu'un autre datagramme rattaché à ce descripteur soit chargé. Il est en effet possible de voir le même datagramme réclamé plusieurs fois de suite.

- **Accès aux réductions d'images**

  Lorsque l'on consulte la base de données grâce à des listes d'images réduites, il ne saurait être question de charger la totalité des réductions en mémoire: à raison de 5Ko par image, des problèmes vont apparaître rapidement. C'est pourquoi il faut charger chaque image au coup par coup. Cependant, lorsque ces images sont chargées, il faut éventuellement les tramer, pour les raisons décrites au chapitre 2.3, et les convertir de manière à ce qu'elles soient compatibles avec le mode d'affichage, ce qui est un processus particulièrement coûteux en temps CPU.

Bien que les réductions d'images soient considérées comme des datagrammes (c.f. 3.2, page 25), et sont donc déjà bufferisées, le problème du tramage et de la conversion demeure. C'est pourquoi les images "prêtes à l'affichage" sont stockées dans un buffer.

Ce buffer est très utile lorsque la machine sur laquelle l'utilisateur travaille est équipée d'une carte vidéo standard (VGA, 256 couleurs), mais est superflu lorsque l'affichage est assuré par une carte graphique équipée d'un accélérateur graphique pour Windows. La conversion pour la compatibilité avec l'affichage est alors effectuée par hardware. De plus ces cartes autorisant généralement le 16 millions de couleurs, le tramage est alors inutile.

- **Autres types de bufferisation**

  Sur une machine disposant de (vraiment) beaucoup de mémoire[1], il est possible de mettre en place un buffer disque très facilement, sans programmation. Il suffit d'installer un gestionnaire de cache disque (SmartDrive, par exemple), en allouant beaucoup de mémoire au cache, pour peu que la taille de ce cache soit plus grande que la taille de la base de données, les performances augmentent alors de manière spectaculaire (c.f. Table 5:).

6.2 Compression

On aura compris que la taille de la base de données pose un problème: un rapide calcul nous indique qu'une base qui contiend 1000 images de taille "moyenne" (réduction: environ 5Ko, histogramme HLS: 4Ko, Textures: 5Ko) prend environ 14Mo, ce qui est un peu élevé.

Mais si on examine de près la nature des datagrammes, on se rend compte que l'histogramme HLS n'est en fait qu'une immense matrice creuse: elle ne contient pratiquement que des zéros. De la même manière, pour peu que l'image originale possède de grandes zones à peu près constantes (ce qui est assez fréquent), il y a des chances pour que la réduction à 16 couleurs correspondante contienne de grandes zones qui elles, seront constantes.

D'où l'idée de compresser les datagrammes: de manière totalement transparente les données sont compressées avant d'être stockées dans la base de données. Cette compression peut se faire selon deux méthodes:

- LZW [2] très efficace en matière de gain de place, mais malheureusement assez lente.
- RLE moins performante que LZW au niveau du gain de place, mais très rapide parce que très simple à implémenter. Sur certaines machines[2], le temps perdu à décompresser les données peut être inférieur au temps gagné en lisant des blocs plus petits. Ce qui a pour conséquence d'accélérer les recherches.

1. Plus que 16Mo en tout cas.
2. Typiquement les machines qui possèdent un CPU rapide et des disque lents (controleur IDE).

Les datagrammes les plus sensibles à cette compression sont les histogrammes HLS qui passent de 4096 à environ 170 octets (pour RLE et LWZ) et les réductions qui gagnent entre 500 octets (RLE) et 1Ko (LZW), les datagrammes "textures" étant constitués de réels, ils sont difficilement compressibles (c.f. Table 2: et Table 3:).

	Pas de compression	Compression RLE	Compression LZW
Taille de la base	14 645 Ko	9 432 Ko	8 881 K0

Table 2: : Compression: performances globales.

	RLE	LZW
Réductions	17%	29%
Datagrammes couleurs	96%	96%
Datagrammes noms	0%	0%
Datagrammes textures	0%	0%

Table 3: : Gain moyen en volume par type de datagrammes.

Le prix à payer d'une telle amélioration est en général un ralentissement des recherches (important pour LZW, faible pour RLE). Il faut décompresser les datagrammes avant de pouvoir les exploiter. Aussi le code permettant de relire les datagrammes, a-t-il été écrit de manière à pouvoir relire des données compressées ou non. De plus une option permettant de spécifier si, lors d'ajout d'images, les datagrammes ajoutés doivent être compressés ou non, est disponible.

6.3 Autres optimisations

Mis à part le buffering et la compression des datagrammes, le reste de l'application n'est pratiquement pas optimisé, sauf en quelques endroits où le temps d'exécution était critique, comme le tramage, où les calculs sont faits en tenant compte de l'architecture 32 bits[1] des machines à base de processeur 386 (ou supérieurs).

Par ailleurs le code est en général écrit avec des astuces simples pour accélérer le traitement, comme utiliser systématiquement des puissances de deux pour les dimensions des tableaux multi-dimensionnels, remplacer les divisions et les multiplications par des puissances de 2 par des shifts, créer des tables à chaque fois que l'on va avoir à utiliser une opération floating-point de manière intense.

1. Tous les accès mémoires sont faits avec des entiers sur 32 bits: très efficace si le code est compilé avec des instructions 386, mais catastrophique avec des instructions 8086 ou 80286.

6.4 Points faibles

Mais le point faible de notre application reste la structure plate de la base de données, cela ne pose aucun problème lorsque la base contient quelques centaines d'images, mais commence à devenir franchement inquiétant lorsque la base en contient plusieurs milliers: le temps d'une recherche peut alors prendre plusieurs dizaines de secondes (c.f. 7.2, page 68).

Par ailleurs, le fait qu'il ne soit pas possible de spécifier un précision quelconque lors d'une recherche par texture peut être dérangeant. Une telle possibilité n'a pas été implémentée car il est très difficile de prévoir avec certitude l'influence de variation des paramêtres statistiques sur une textures.

Enfin une évaluation rigoureuse des résultats de recherche manque. Il eut été possible de procéder de différentes manières:

- Pour la recherche par couleur, en tentant de retrouver des images se trouvant dans la base à partir d'une version imprimée de son contenu. Puis en compatibilisant les recherche fructueuses. Une telle méthode donnerait plutôt une mesure qualitative quand à l'efficacité du logiciel.

- Toujours pour la recherche par couleur, avec un paramètre de recherche déterminé. Il serait possible de comparer les résultats de la recherche avec des mesures faites directement sur les images originales. Ce type de test, très long, permettrait d'obtenir des résultats quantitatifs.

- Les recherches par textures seraient plus difficiles à tester, il est probable que seules des mesures qualitatives telles celles décrites auparavant ne soient facilement réalisables.

6.5 Mesures

Quelques mesures ont été effectuées sur deux machines: une première, déjà ancienne et une seconde, pratiquement à la pointe de ce qui se fait au moment de la rédaction de ce rapport:

- un 486-33
  8 Mo RAM
  Contrôleur disques IDE
  Vidéo S-VGA (800x600x256 couleurs), pas d'accélérateur graphique.

- un Pentium 90
  32 Mo RAM
  Contrôleur disques SCSI-II[1]
  Vidéo ATI (1280x1024x16Millions couleurs). accélérateur graphique.

Les tests illustrés aux tableaux ont été faits dans une base de données qui contient 1131 images.

1. Les contrôleurs SCSI offrent de bien meilleurs taux de transfert que les contrôleurs IDE.

Recherche / Base	pas de compression	compression RLE	compression LZW
Couleurs	7 sec	7 sec	43 sec
Noms	5 sec	5 sec	5 sec
Textures	7 sec	7 sec	7 sec

Table 4: : Recherches: Pentium, pas de cache disque.

Recherche / Base	pas de compression	compression RLE	compression LZW
Couleurs	4 sec	4 sec	40 sec
Noms	3 sec	3 sec	3 sec
Textures	4 sec	4 sec	4 sec

Table 5: : Recherches: Pentium, cache disque 16 Mo.

Recherche / base	pas de compression	compression RLE	compression LZW
Couleurs	19	15 sec	200 sec
Noms	13	12 sec	12 sec
Textures	23	22 sec	22 sec

Table 6: : Recherches: 486, pas de cache disque.

On remarque que, quelque soit le type de compression utilisé, le temps de recherche par noms ou par textures ne varie pas,ou très peu. Ce qui est naturel: SPIP stocke de manière non compressée les datagrammes qui prendraient plus de place s'ils étaient compactés. Ce qui est typiquement le cas des datagrammes noms (8 caractères) et des datagrammes textures (ensemble de réels).

Par ailleurs, il est à noter que la compression RLE donne de meilleurs résultats tant en place disque, notament sur le 486, qu'en temps d'exécution comme cela avait été prévu au paragraphe 6.2.

Par ailleurs, on remarque l'excellent gain de performance qu'offre un cache disque important. L'expérience n'a pu être effectuée sur un 486, par manque de machine adéquate.

7 Conclusion et perspectives

Comme cela avait été prévu, les bases d'une application permettant de stocker des images dans une base de données pour ensuite les retrouver en fonction de leur contenu ont été posées. Par ailleurs ce logiciel, contrairement aux produits issus de laboratoires de recherche, présente l'avantage de tourner sur une machine grand public. De plus, comme on a pu le voir, il est très évolutif, ce qui, dans un milieu commercial représenterai un avantage certain sur la concurrence.

Cependant, de nombreuses améliorations restent à effectuer. Ajouter d'autres recherches bien sûr: recherche en fonction de primitives géométriques ([4] [16]), recherche en fonction de l'allure générale d'une image référence [15] etc. Mais rajouter des paramètres de recherche n'est pas tout.

7.1 Optimisations

De nombreuses optimisations sont encore possibles pour augmenter l'efficacité de SPIP. On distingue en particulier:

- **Les boucles de traitement**

  La plupart des boucles de traitement d'images sont constituées de deux boucles imbriquées, qui englobent quelques lignes de code portant sur du calcul sur des entiers. Celui qui s'en sentira le courage, pourra réécrire ces lignes en assembleur, ce qui permettrait de gagner un facteur de temps considérable, surtout si l'on écrit ce code de manière à tenir compte de l'architecture super-scalaire du Pentium [12].

- **Décompression**

La routine de décompression LZW avait été à l'origine développée pour la décompression des fichiers TIFF, son optimisation était alors importante, certe, mais pas primordiale. Etant donné qu'elle est désormais utilisée pour décompresser les datagrammes, il serait judicieux de l'optimiser au maximum, voir de la réécrire en assembleur.

7.2 Créer une arborescence dans la base

On a vu que le plus gros défaut de notre application était cette structure plate de la base de données, qui donnait des résultats satisfaisants lorsque la base contenait quelques centaines d'images, mais perd rapidement son efficacité lorsque la base occupe plusieurs milliers d'images.

La solution consiste à créer une arborescence à l'intérieur de la base, ce qui permettrait d'accélérer la recherche dans la base de données. Mais notons qu'un problème va se poser. On a vu que l'architecture ouverte de l'application interdisait à la partie qui gère la base de données de connaître la signification du contenu des datagrammes (c.f. 3.2, page 25); cependant quelque soit le type d'arborescence employée, elle devra être construite en fonction du contenu des datagrammes. Par conséquent, le code qui construira cette arborescence (qui fera partie du code qui gère la base de données) devra collaborer étroitement avec chaque module de recherche, ce qui ne se fera pas sans peine!

La plupart des bases de données d'images, tels QBIC utilisent une arborescence nommée R-tree pour stocker leurs données [19],[20]. Les R-trees se présentent sous la forme d'un arbre dont les noeuds terminaux (les feuilles) sont des éléments de la base. Les noeuds intermédiaires sont des de clusters (régions) dans l'espace de recherche qui englobent les feuilles. Cette méthode permet d'accélérer considérablement les recherches, en les limitant aux éléments se trouvant dans certains clusters: ceux qui correspondent aux paramètres de recherche. Notons que ces clusters peuvent se chevaucher, ce qui autorise les éléments à se trouver dans plusieurs d'entre eux.

Une autre arborescence intéressante est constituée par une variable de R-trees, les TV-trees [17], qui permettent de stocker des éléments différents (i.e. de dimensions différentes), mais qui présentent tout de même un petit inconvénient: ils ne travaillent qu'avec des scalaires ou des vecteurs composés de scalaires, ce qui signifie qu'il faudra très certainement trouver le moyen de résumer les datagrammes, qui ne sont pas à base de vecteurs comme les histogrammes par couleurs, sous une forme qui le sera.

Notons par ailleurs, que l'ajout d'une image dans la base sera considérablement compliqué, il faudra probablement modifier toute l'arborescence à chaque ajout. C'est entre autre pour cette raison que QBIC ne gère que des bases de données statiques (construites une bonne fois pour toutes à partir d'un CD-ROM par exemple).

7.3 DLL

On a vu que le code, qui régissait les paramètres de recherche, avait été volontairement isolé afin de faciliter l'ajout de recherches supplémentaires par la suite. L'étape suivante consiste à encapsuler ce code dans des DLL (Dynamic Link Library) [13]. Ce qui permettrait de rendre les modules de recherche totalement indépendants du code de la base de données. Il ne serait alors plus nécessaire de faire appel aux services d'un compilateur pour ajouter un paramètre de recherche.

A long terme, si SPIP doit être diffusé, même de façon restreinte, il deviendra impossible de gérer un nombre x (très grand) de versions différentes de l'application, ne différant que par leurs possibilités en matière de recherche. Alors qu'avec ce système, il suffit de fournir une seule fois l'application principale, et ensuite les DLL nécessaires.

ANNEXES

Références

[1] “Image File Survey Results”, Advenced imaging, june 1994, pp 10.

[2] C. Wayne Brown, Barry J. Shepherd, “Graphics File Format, reference and guide”, Manning Publication Co, 1995.

[3] D. Travis, “Effective Color Display, theory and practice”, Academic Press, 1991.

[4] C. Faloustsos, M. Flickner, W. Niblack, D. Petkovic, W. Equitz, R. Barber; “Efficient and Effective Querying by Image Content”, IBM. Research report, Research division of San Jose, 1993.

[5] M. Miyahara, Y. Yoshida “Mathematical transform of (R,G,B) color data to Munsell (H,V,C) color datas”, SPIE Vol. 1001 Visual Communication and Image Processing, 1988.

[6] B.M. Mehtre, M. S. Kankanhalli, A.D. Narasimhalu, G. C. Man. “Color matching for image retrieval”, Pattern Recognition Letters, March 95, pp 325-331.

[7] C. Rauber; “La transformée de Gabor et ses applications”, Travail de diplôme, CUI, 1989-90.

[8] H. Tamura, S.Mori, T. Yamawaki, “Textural features Correspondinf to visual Perception”, IEEE Transaction on Systems, Man and Cybernetics. Vol. SMC-8 N°6, June 1976.

[9] F. Tomita, S. Tsuji, “Computer Analysis of Visual Textures”, Kluwer Academic Publishers, 1990.

[10] T. Pun, “Cours d’Imagerie Numérique”, Université de Genève, 1991, 1992, 1993.

[11] J.D. Foley, A. Van Dam, S.F. Feiner, J.F. Hughes, “Computer Graphics: Principles and Practice”, Second edition. Addison-Weslwy, 1990.

[12] M. Tischer, “La Bible PC, Programmation Système”, Quatrième édition Micro Application, 1993.

[13] C. Petzold, “ Programming Windows”, second edition, Microsoft Press, 1990.

[14] W. Niblack, M. Flickmner “ Find me the pictures look this: IBM’s Image Query Project”, Advenced Imaging, April 1993.

[15] T. Kato, T.Kurita, N.Otsu, K. Hirata, “A Sketch Retrival for full Color Image Database”, IEEE (?) 1992.

[16] W.F. Cody, L.M. Haas, W.Niblack; “Querying Multimedia Data from Multiple Repositories by Content: the Garlic Project”, IFIP 2.6, Third Working Conference on Visual Database Systems, EPFL, 1995.

[17] KinIp Lin, H.V. Jagadish, C. Faloutsos. “The TV-Tree: An Index Structure for High-Dimensional data”, VLDB journal, 3, pp 517-542, 1994.

[18] A. Jacot-Descombes, M.Rupp, T.Pun, “LaboImage: a portable windows-based environnement for research in image processing and analysis”, SPIE Vol. 1659 Image processing and Interchange, 1992.

[19] M. A. Kang, K. Li, “Query Processing Methods for Connectivy Search in Visual Database Using R+-tree”, IFIP 2.6, Visual Database Systems, EPFL, 1994, pp 365-377.

[20] N. Roussopoulos, C. Faloutsos, T. Sellis “An Efficient Pictorial Database System for PSQL”, IEEE transactions on software engineering, vol.14, No 5, May 88, pp 639-650.

Inventaire

Avec ce travail sont fournis:

- Un rapport théorique
  Que vous tenez entre les mains.

- Un rapport technique
  Relate les détails d'implémentation de SPIP, un survol de ce document pourrait est si on désire mettre à jour SPIP.

- 2 cassettes video
  La première au format VHS et la seconde au format BetaCam (bien meilleure qualité). Toutes deux contiennent une démo rapide (12 minutes) des possibilités de SPIP.

- 11 disquettes.
  Deux jeux de 4 disquette permettent d'installer SPIP sur un PC (nécessite au minimum Windows 3.1). Attention! un de ces jeux de disquettes permet également d'installer les sources complets de l'application.

  Trois disquettes contenant les outils utilisés pour développer SPIP.