



**Centre de recherche
informatique de Montréal**

550, rue Sherbrooke Ouest, bureau 100
Montréal (Québec) H3A 1B9
Téléphone : (514) 840-1234
Télécopieur : (514) 840-1244
<http://www.crim.ca>

CRIM - Documentation/Communications

**Rapport technique
Java, JAI et code natif
en vision artificielle et traitement d'image**

Première version

CRIM-01/01-12

Marc Lalonde
Agent de recherche senior
RECO

23 janvier 2001

Collection scientifique et technique

ISBN 2-89522-010-7

Pour tout renseignement, communiquer avec:
CRIM Centre de documentation
Centre de recherche informatique de Montréal (CRIM)
550, rue Sherbrooke Ouest, bureau 100
Montréal (Québec) H3A 1B9

Téléphone : (514) 840-1234
Télécopieur : (514) 840-1244

Tous droits réservés © 2001 CRIM
Bibliothèque nationale du Québec
Bibliothèque nationale du Canada
ISBN 2-89522-010-7

TABLE DES MATIÈRES

INTRODUCTION.....	4
1. JAVA ET JAI	4
2. JAVA NATIVE INTERFACE (JNI).....	6
3. MANIPULATION DE PIXELS AVEC FONCTIONS JNI.....	8
4. PERFORMANCE	10
CONCLUSION.....	11

Introduction

Ce document présente des tests effectués pour évaluer l'intérêt d'intégrer du code C de vision artificielle à Java de manière flexible et efficace. Le but visé est de voir comment il serait possible de faire rouler des algorithmes écrits en C qui manipuleraient des images créées et gérées par Java. On pourrait ainsi exploiter les avantages apportés par Java et surtout son extension JAI (Java Advanced Imaging) sans pertes notables de performance.

La version Java utilisée est le Java 2 Platform, Standard Edition, version 1.3, avec JAI version 1.1 beta.

1. Java et JAI

JAI est une extension de Java conçue pour des applications de pointe en traitement d'image, en particulier les applications distribuées. En plus des fonctionnalités de base (chargement/sauvegarde d'images, opérateurs variés de traitement d'image, etc.), JAI permet de traiter des images à distance, de manipuler des images indépendamment de leur format ou de leur dimension, de gérer des images de grand format par le découpage en tuiles, de créer de nouveaux opérateurs, etc. Dans un premier temps, l'intérêt se limite aux structures de données (classes d'images) puisque les autres services de JAI sont ou bien inutiles ou bien semblables à d'autres services offerts p. ex. par la bibliothèque de CVIPtools.

Plusieurs classes d'images sont définies dans JAI. La classe (abstraite) qui donne naissance à toutes les autres est `PlanarImage`, qui implémente notamment les méthodes de l'interface `RenderedImage` définie dans `java.awt.image`. Les classes dérivées de `PlanarImage` qui semblent les plus utiles sont `TiledImage` et `RenderedOp`. `RenderedOp` est la classe d'image produite à la suite de l'application d'un opérateur. C'est un élément de base au "framework" distribué de JAI qui permet de créer des chaînes d'opérations où chaque élément de la chaîne mémorise l'opération à effectuer, les paramètres requis et l'image produite. Un usager peut donc construire une "application", modifier les paramètres ou la séquence d'opérations, et finalement demander l'activation de la procédure ("rendering"). Cet aspect n'est pas couvert ici. `TiledImage` est la seule classe permettant de modifier des pixels. En fait, toute image stocke les pixels dans un objet de type `Raster` ou `WritableRaster` (classe dérivée de `Raster`); un objet `Raster` possède des fonctions du genre `getPixel` ou `getSample` pour lire des valeurs de pixels, mais c'est `WritableRaster` qui possède en plus des fonctions `setPixel` et `setSample` nécessaires afin de modifier des pixels. `Raster` encapsule une classe (abstraite) `DataBuffer` qui est la classe la plus près des pixels et de leur organisation. En effet, les sous-classes de `DataBuffer` (créées pour différents types de

dimension de pixels, p. ex. `DataBufferByte`, `DataBufferInt`, etc.) possèdent comme donnée membre un tableau 1D qui stocke les pixels de l'image.

Le traitement d'une image au niveau des pixels peut se faire de plusieurs façons. Des itérateurs peuvent être employés pour passer d'un pixel à l'autre, d'une ligne à l'autre, d'une bande à l'autre:

```
RectIter iter=RectIterFactory.create(img,null);
iter.startBands();
while (!iter.finishedBands()) {
    iter.startLines();
    while (!iter.finishedLines()) {
        iter.startPixels();
        while (!iter.finishedPixels()) {
            pixel=iter.getSample();
            ...
            iter.nextPixel();
        }
        iter.nextLine();
    }
    iter.nextBand();
}
```

L'intérêt des itérateurs est qu'une région d'intérêt (une zone rectangulaire dans l'image) peut être défini et les frontières de la région seront respectées par les itérateurs sans que l'utilisateur ait à gérer les limites.

On peut aussi extraire l'objet `Raster` de l'image et se servir des fonctions membres `getPixel()` ou `getSample()`. De même, on peut extraire l'objet `DataBuffer` de l'objet `Raster` et se servir de sa fonction `getElem()`. Une autre possibilité est d'invoquer une méthode `getData` définie pour p. ex. un objet `DataBufferByte`, qui retourne carrément le tableau de pixels (sous forme de type de base, p. ex. `byte` ou `int`) qu'on peut modifier à notre guise. Rappelons qu'on ne peut pas avoir d'objet `DataBuffer` puisque cette classe est abstraite. Finalement, il est possible de lire des tuiles de l'image à l'aide de la méthode `getTile` disponible pour un objet de type `PlanarImage` (ou une classe dérivée), laquelle retourne un objet `Raster` qu'on manipule à l'aide des méthodes appropriées :

```
Raster rst;
destbuf=destraster.getDataBuffer();

int tiloffset=0;
int th,tw; // hauteur, largeur d'une tuile
for(int tily=1; tily<srcimg.getNumYTiles(); tily++)
    for(int tilx=0; tilx<srcimg.getNumXTiles(); tilx++)
    {
        rst=srcimg.getTile(tilx,tily);
        th=rst.getHeight();
    }
```

```
tw=rst.getWidth();
int roffsetx=rst.getMinX(); // Les coord. d'un pt dans une tuile
int roffsety=rst.getMinY(); // sont relatives à l'image globale...

for(int y=0; y<th; y++)
  for(int x=0; x<tw; x++)
    for(int b=rst.getNumBands()-1; b>=0; b--, tiloffset++)
      destbuf.setElem(tiloffset,
        rst.getSample(roffsetx+x,roffsety+y,b)/2);
} // for
```

Ce code copie simplement les pixels d'une image source dans le Raster d'une image de destination (en divisant l'intensité par deux). Un peu de gestion est nécessaire pour le traitement par tuiles étant donné qu'il ne faut pas se fier à la procédure de découpage de Java : par exemple, une image RGB somme toute petite (640x480) a été découpée en 120 tuiles verticales par Java au moment de son chargement (une seule tuile en horizontal).

Puisqu'il existe plusieurs manières d'accéder aux pixels d'une image, la question est maintenant de savoir comment exploiter ces manières à partir d'une fonction C.

2. Java Native Interface (JNI)

JNI définit le "protocole" à suivre pour pouvoir appeler, à partir de Java, des fonctions native provenant d'une bibliothèque partagée (.dll ou .so). Résumé simplement, il permet p. ex. à une fonction C de recevoir en paramètre des objets Java (des types de base, des tableaux, des instances de classes, etc.), d'invoquer leurs méthodes, d'en créer d'autres, etc. Une sorte de dualité est permise par JNI qui fait que plusieurs actions faites en Java peuvent aussi être faites par une fonction native. Brièvement, voici comment une fonction native f(...) est déclarée et construite:

Dans Java, f(...) est une méthode d'une classe, p. ex. F:

```
public class F {
    public native void f(Raster rasterin, Raster rasterout);
    static {
        System.loadLibrary("NativeTest");
    }
    ...
}
```

Le bout de code "static { ... }" sert à charger la bibliothèque NativeTest (fichier Native-Test.so sur Unix) contenant la fonction C f().

- Le fichier Java doit être compilé :

```
$ javac F.java
```

- Un "header file" F.h contenant le prototype de la fonction à coder doit être créé pour le fichier C:

```
$ javah -jni F
```

- f(...) a une signature du type

```
JNIEXPORT void JNICALL Java_F_f (JNIEnv *, jclass, jobject, jobject);
```

où les deux premiers paramètres sont là pour permettre aux fonctions JNI de faire le lien avec la machine virtuelle Java et où les deux derniers paramètres sont les objets Java tels que définis dans la classe F.

- Il ne reste plus qu'à coder la fonction Java_F_f, à compiler, à construire la bibliothèque partagée (sur Unix: `ld -G -o NativeTest.so toto.o`) et à s'assurer que la bibliothèque créée est accessible, i.e. dans un chemin pointé par `LD_LIBRARY_PATH`.

Les types de base Java comme `int`, `char`, `byte`, etc. ont un pendant C: `jint`, `jchar`, `jbyte`, etc. Les types complexes (des objets) sont plus difficiles à utiliser puisqu'il faut suivre une certaine procédure. Par exemple, si notre fonction `f` est déclarée comme:

```
void Java_F_f( JNIEnv *env, jclass cls, jobject obj1, jobject obj2)
```

et qu'on veuille invoquer la méthode "float getFoo(int,int)" de l'objet `obj1`, on doit écrire le code suivant:

```
jclass obj1cls;
jmethodID mid;
float floatdata;

...

obj1cls = (*env)->GetObjectClass(env, obj1); // obtenir la classe de obj1
mid = (*env)->GetMethodID(env, obj1cls, "getFoo", "(II)F");
if (mid == 0) {
    return;
}
floatdata = (*env)->CallFloatMethod(env, obj1, mid, IntArg1, IntArg2);
```

La fonction "GetMethodID" trouve l'identificateur de la méthode voulue et pour ce faire requiert la signature de cette méthode ainsi que le type de retour: c'est la chaîne "(II)F", qui spécifie que la méthode recherchée accepte deux `int` en paramètres et retourne un `float`. La fonction `CallFloatMethod` invoque la méthode (avec les deux paramètres `IntArg1` et `IntArg2`). Naturellement, des méthodes avec des signatures beaucoup plus com-

plexes peuvent être invoquées (p. ex. fonction qui reçoit un objet en paramètre et qui retourne un tableau de Strings). L'utilitaire `javap` est particulièrement utile pour trouver ces chaînes de signatures. On n'a qu'à taper au shell:

```
$ javap -s <nom de la classe>
```

et on obtient la signature de chaque méthode. Par exemple, si on veut la signature de la méthode qui copie un Raster dans un objet TiledImage, on tape:

```
$javap -s javax.media.jai.TiledImage  
et on peut voir entre autres :
```

```
public void setData(java.awt.image.Raster);  
  
/* (Ljava/awt/image/Raster;)V */
```

où la lettre L de la signature indique que l'expression qui suit est une classe. Il faut s'assurer que la signature est bonne, autrement Java génère une exception "NoSuchMethod" pendant l'exécution.

Plusieurs fonctions JNI sont disponibles pour faire plein de choses; le lecteur est renvoyé à la documentation de JNI pour plus de détails (ou à Liang (1999) donné en référence).

3. Manipulation de pixels avec fonctions JNI

Plusieurs tests ont été faits pour accéder aux pixels d'une image Java à partir d'une fonction native C. Le problème, simple, était toujours le même: copier une image RGB dans une autre, en modifiant chaque pixel au passage (p. ex. division de sa valeur par 2).

En première approche (la plus intuitive), les objets Raster des images d'entrée et de sortie ont été passés à la fonction native et leurs méthodes `getSample` / `setSample` ont été utilisées:

```
for(x=0; x<width; x++)  
  for(y=0; y<height; y++) {  
    pix=(*env)-CallIntMethod(...) // avec ID de la méthode getSample  
    (*env)-CallVoidMethod(..., pix) // avec ID de la méthode setSample  
  }
```

Les résultats ont été catastrophiques. Alors qu'en Java l'opération prend environ 3-4 secondes, la fonction native faisait le travail en 30 secondes! D'autres variantes ont donné les mêmes résultats (p. ex. utiliser les méthodes `getElem` et `setElem` de `DataBuffer` (de sa classe dérivée en fait)). La conclusion est qu'il est très coûteux d'invoquer des méthodes de classes Java à partir d'une fonction native, à plus forte raison en traitement d'image où

des centaines de milliers d'appels doivent être faits lorsqu'on parcourt une image. D'ailleurs, Liang (1999) souligne que ce type d'appel "natif/Java" est peu optimisé et qu'il peut être jusqu'à 10 fois plus lent qu'un appel "Java/Java". La pratique lui donne raison.

L'approche à prendre semble plutôt de tenter d'accéder directement au tableau de pixels en utilisant des fonctions JNI pour obtenir un pointeur C sur le tableau. Si des objets Raster sont passés en paramètres à la fonction native, ceci implique qu'une cascade d'opérations doit être faite pour extraire la classe de chaque objet Raster, son objet DataBuffer, la classe de l'objet DataBuffer et finalement son objet tableau. Un test a été fait avec cette approche et la performance est naturellement excellente, pourvu que le traitement à faire soit substantiel (autrement l'*overhead* provoqué par cette cascade d'opérations devient significatif). Qui plus est, du code C a été écrit pour alléger le plus possible l'aspect transformation de données et on obtient le bout de code suivant, pas mal semblable au code écrit avec la bibliothèque CVIPtools:

```
{
  int width,height;
  int i, size;
  int nband, x, y, minx, miny;
  unsigned char pixel;
  Image *inim, *outim;

  printf("native started.\n");
  inim=buildImageFromJavaRaster(env,rasterinobj);
  outim=buildImageFromJavaRaster(env,rasteroutobj);

  for(nband=0; nband<getNoOfBands_Image(inim); nband++)
    for(x=0; x<getNoOfCols_Image(inim); x++)
      for(y=0; y<getNoOfRows_Image(inim); y++)
        setPixel_Image(outim,byte,x,y,nband,
          getPixel_Image(inim,byte,x,y,nband));

  printf("native ended.\n");

  (*env)->ReleaseByteArrayElements(env,inim-rasterarray,
    (jbyte*)inim-raster,0);
  (*env)->ReleaseByteArrayElements(env,outim-rasterarray,
    (jbyte*)outim-raster,0);
}
```

Le type Image a été réécrit pour l'occasion, ainsi que les macros `getNoOfBands_Image`, `getNoOfRows_Image`, `getNoOfCols_Image`, `setPixel_Image` et `getPixel_Image`. Une fonction `buildImageFromJavaRaster` a aussi été écrite pour créer une structure Image à partir d'un objet Raster (l'*overhead* est minimal, il n'y a pas de copie de pixels;).

Cet exemple laisse croire que le pont entre Java et du code écrit p. ex. avec la bibliothèque CVIPtools pourrait se faire facilement tant que des macros du type `getNoOfXXX_Image()` sont utilisées.

4. Performance

Quelques tests simples de performance ont été réalisés pour le même problème de copie d'image RGB décrit plus haut. Un programme écrit en C fait l'opération en 0.25 secondes environ, incluant le chargement de l'exécutable et le chargement de l'image RGB (922k). Le temps mesuré est le temps de CPU total ("wall clock time"), tel que rapporté par la commande `time (csh)`, sur Solaris 7 x86, processeur Pentium II, 350 MHz. Il est évidemment approximatif, puisqu'il dépend de la charge ("load") de la machine qui exécute le programme.

Pour un programme Java, le temps de chargement de la machine virtuelle, du programme et de l'image (sans traitement) est environ 3.9 secondes dans les mêmes conditions.

- Une fois rajoutée la copie d'image à l'aide des méthode `getElem / setElem (DataBufferByte)`, le temps total grimpe à 5.3 secondes environ. La copie de 640*480*3 pixels prend donc environ 2 secondes.
- Si on extrait les tableaux de pixels des `DataBuffer` source et destination et qu'on copie un tableau dans l'autre, élément par élément, le temps total est d'environ 5.2 secondes.
- L'approche par tuiles donne aussi un temps de traitement de 5.2 secondes.
- Comme on peut s'y attendre, la version avec itérateurs coûte cher: 8.5 secondes.

Pour l'approche native, un coût supplémentaire doit être assumé puisque la bibliothèque partagée contenant le code natif doit être localisée et chargée. C'est naturellement un coût fixe, au même titre que le chargement des classes Java. Par ailleurs, de façon générale, les opérations Java coûtent cher: même si une fonction native est appelée, ce sont des méthodes Java qui sont utilisées pour p. ex. créer l'image de destination avec un "SampleModel" et un "ColorModel" appropriés; or ces opérations prennent quand même 0.4 secondes à être effectuées (forcément, ce temps se retrouve aussi inclus dans le temps total donné pour la version Java pure). Dans la même veines, comme la fonction native telle que présentée à la section précédente requiert des objets `Raster` en argument, le temps de création de ces objets est non négligeable: environ 0.75 seconde. Le temps d'exécution de la fonction native est estimé à environ 0.3 seconde, probablement à cause de l'invocation de la cascade d'appels aux fonctions JNI. On arrive à un temps total d'environ 5.25 secondes, qui est virtuellement identique au temps total de la version Java pure. Un ré-

examen des performances de la version Java pure montre que le temps pris pour effectuer la copie de pixels proprement dite est d'environ 0.3 secondes.

L'exemple est trop simple pour avantager la version native; une tâche plus complexe que la simple copie de pixels (p. ex. une convolution) serait beaucoup plus discriminante. Cependant, il est intéressant de voir la "ventilation" des temps d'exécution, en particulier pour l'extraction de l'objet Raster d'une image; une autre approche est peut-être possible, mais à première vue, c'est l'unique moyen d'atteindre les pixels directement sans passer par les itérateurs. L'impression qui se dégage est qu'une application voulant exploiter la vitesse du code natif devrait réduire au minimum la manipulation d'objets de haut niveau (p. ex. `PlanarImage` ou `Raster`) et définir des classes d'images alternatives, légères et faciles d'accès à partir du code natif (i.e. pour lesquelles les pixels sont arrangés directement dans des tableaux).

Conclusion

Intuitivement, l'emploi de code natif pour accélérer le traitement à faire sur des images est possible et même souhaitable: même JAI le fait puisque plusieurs fonctions de JAI sont disponibles sous forme native et que ce sont elles qui sont privilégiées si la bibliothèque partagée qui les regroupe est disponible sur la plate-forme utilisée (pour l'instant, seuls une DLL pour Microsoft Windows et un `.so` pour Solaris sont disponibles); autrement, c'est l'implémentation Java qui est utilisée. Cependant, la séparation entre Java et code natif doit être telle que certaines situations devraient être évitées : des fonctions natives petites et peu complexes (aucun avantage de performance), des fonctions natives qui reçoivent des objets complexes (l'accès aux données de ces objets est coûteux à partir d'une fonction native) ou même des objets complexes du côté Java qui doivent être décomposés en objets « atomiques » prêts à être passés à du code natif. Pour l'utilisation de Java en vision, ceci veut probablement dire une redéfinition de la classe image afin de faciliter l'accès aux pixels (p. ex. en gardant une référence à un objet `Raster` ou `DataBuffer` et en favorisant le traitement sur ce type d'objet, plutôt que sur les objets de plus haut niveau comme `PlanarImage`).

Références

S. Liang. "The Java Native Interface; Programmer's Guide and Specification". Addison-Wesley, 1999.

Java 2 Platform, Standard Edition, v 1.3. API Specification.
<http://java.sun.com/j2se/1.3/docs/api/index.html>

Java Advanced Imaging API 1.1 Documentation, disponible à
<http://java.sun.com/products/java-media/jai/docs/index.html>

Java Advanced Imaging Programming guide (v 1.0.1), disponible à
<http://java.sun.com/products/java-media/jai/docs/index.html>

B. Stearns. « Trail : Java Native Interface ». Disponible à
<http://java.sun.com/docs/books/tutorial/native1.1/index.html>