Developing a raster detector system with the J array processing language

by Jan Jacobs

All digital copying aims to reproduce an original image as faithfully as possible under certain constraints. In the past, image processing had to be implemented in hardware for performance reasons. Here, a 100% software solution is outlined. In order to find such a solution an appropriate methodology based on the array processing language J is used. Although J is ideal for prototyping such designs, its wider application is seriously hindered by the lack of awareness of array processing languages amongst engineers, and by the lack of available education in this language and methodology.

Digital copying involves a process called half-toning, which transforms a continuous range of tones into a two-tone image. Text and image have different half-toning procedures, and a relative simple edge detector can differentiate between them. However, raster images confuse detectors by letting them think erroneously that text instead of image has to be half-toned. The consequence of this is shown by the 'lighter' blobs in the clipped halftone result (shown by a rectangle in Fig. 1).

This article describes the development of a local raster detector to solve this problem, using a proven methodology using the relatively new array processing language J.

Towards system requirements

In principle it is possible to detect the existence of a raster in a local neighborhood by, for example, using a Fourier transform. However, developing a detector on a standard PC platform requires that adequate quality is achieved with a computation time for detection, which does not exceed 0.5s for an A4 original scanned at 300 dpi (dots per inch, a measure for printing resolution) resolution. Also the detection should be independent of the raster form (e.g. line, dot rasters) and orientation. It should also be based on information retrieved locally, and it has been determined that 8×8 pixel sized tiles should be adequate. Since only grey level images are considered, pixels may be taken as equivalent to bytes.

The frequency band causing half-toning problems is 60-135 lpi (lines per inch, a measure for raster frequency), and another problem is caused by the erroneous detection of text as a low-frequency raster. It has been empirically
The methodology supports a traditional incremental design process but with one difference, namely that intermediate designs are executable.

For the requirements phase this means that early feedback is possible and that functional specifications can be agreed on in an early stage (see Fig. 2, where the executable specs are denoted by 'architecture'). The actual architecture for rasterDetect will be derived later.

For the implementation phase early feedback on design alternatives and (cumulative) design choices minimise proliferation of errors. A first implementation is outlined later (see also Fig. 2).

In the development process, choices are recorded for both phases. This is done in such way that, at all major decision points, choices can easily be undone to support changes in design.

The approach is based on the 'computer architecture' methodology for hardware and software systems developed by Blaauw and Brooks while working at IBM in the late 1960s. At that time the language APL was used to support the methodology.

Here the successor language J is used to support both the problem analysis and the design process. J fragments will be given that helped in developing the architecture as well as the implementation process in an interactive way. The code given in these fragments is accompanied by examples. The reader should note that, because of the Journal column widths, the J fragments have been 'wrapped around' and would normally appear on one line.

In summary, a solution has to be found which meets the following requirements:

- Computation time must be under 0.5s on a 600MHz Pentium III processor.
- Only 60-135 lpi rasters, independent of raster form and orientation, should be detected.
- Text should not be classified as a raster.
- Strict horizontal or vertical line rasters should not be classified as a raster.

In this phase the required functionality of the detector is developed. It is coded in an executable specification (architecture) without the burden of implementation constraints. The analysis as well as the construction of the architecture is supported by J code fragments.

Promising methods found in the literature to detect rasters are often based on Fourier transforms to solve the following problems:

- independence of raster form (dots, line pairs etc.)
- independence of raster orientation (all angles)
- handling of text
• handling of strict horizontal or vertical line rasters.

Execution speed is not an issue in this phase.

To start the investigation, a part of the problem area in Fig. 1 is read in a matrix.

```
load 'graph' NB. load image library
bitmap =: readbmp 'raster.bmp' NB. read bitmap
```

A small bitmap (16 x 16) is read into matrix variable bitmap (Fig. 3).

```
load 'addons\fftw\fftw'
NB. load fft library
tiles =: (2 2$8) <.;.-3 bitmap
NB. tessellate bitmap
part at the right-hand side.
tilesF =: fftw &.> tiles
```

The verb <.;._3 is a shorthand in J for 'tessellate' with the tiling pattern given by the left argument. This tiling pattern, defined by 2 2$8, evaluates into a 2x2 shaped matrix containing all 8s, is given below:

```
8 8
8 8
```

This matrix actually defines a movement vector and a displacement vector for the two dimensions used in the tiling process. The result of this tessellation is given in Fig. 4.

The spectrum of one of these patches or tiles (upper left) is shown in Fig. 5. The following observations can be made:

- The (amplitude) spectrum contains complex numbers, the real part is at the js left-hand side, the imaginary part at the right-hand side.
- The DC component, upper left corner at (0,0), is relatively large.
- The maximum amplitude can be found at (y, x) = (4, 3) and at (4, 5) indicating a relatively high frequency raster.
- Note the symmetry around column and row number 4, the principal components may be confined into an upper left 5 x 5 sub-matrix.

Analysis indicates that in the problem regions two dominant rasters occur in the 60-135 \( \pi \) band. The solution of the problem is a band detector, which should have the circular symmetry property, since rasters may be positioned under arbitrary angle (see Fig. 6).

```
Fig. 4 Result of tessellation
```

```
Fig. 5 Spectrum of one of the patches or tiles
```
The requested band detector is to work towards a realisation without compromising or altering the functionality (architecture). In this section only a few implementation problems are addressed.

Fig. 6 Requested band detector

<table>
<thead>
<tr>
<th>0</th>
<th>37</th>
<th>75</th>
<th>112</th>
<th>150</th>
</tr>
</thead>
<tbody>
<tr>
<td>37</td>
<td>53</td>
<td>83</td>
<td>118</td>
<td>154</td>
</tr>
<tr>
<td>75</td>
<td>83</td>
<td>106</td>
<td>135</td>
<td>167</td>
</tr>
<tr>
<td>112</td>
<td>118</td>
<td>135</td>
<td>159</td>
<td>187</td>
</tr>
<tr>
<td>150</td>
<td>154</td>
<td>167</td>
<td>187</td>
<td>212</td>
</tr>
</tbody>
</table>

Fig. 7 Band detector approximation (numbers indicate lpi)

The specification for raster detection in the 60-135 lpi interval, resulting in a single boolean per tile, is now derived and expressed in J fragments.

The matrix singleTileF is one of the many frequency domain tiles (tilesF; see Fig. 5). The power spectrum of this 8x8 tile is taken by first squaring each complex number (*) followed by taking its magnitude (|). Note the right to left evaluation order, see Fig. 8. The identity function | forces a display of its right argument (assignments are silent).

\[
\text{powerTileF} = : | *: \text{singleTileF}
\]

Since only the shaded components in Fig. 7 are required we only select the lower 4x4 sub-matrix (4 4[, see Fig. 9):

\[
\text{intermdF} = : 4 4[. \text{powerTileF}]
\]

Looking closer to the components which have to be selected one can cover the shaded area by masking of the lower 2x2 sub-matrix and component (3,3). This is done by function amend (\(0\)), which can update specified (group) components. The left 0 is the update value (see Fig. 10 for the filtered result):

\[
\text{filteredF} = : 0(:,<0,1;0,1;3,3); 4 4[, |*: \text{singleTileF}]
\]

The accumulated power in this frequency band is determined by counting all columns by function +/ into four sub-totals and again +/ to accumulate all column totals. Comparing it with a certain threshold returns the answer whether a raster is present or not:

\[
\text{rasterDetected} =: \text{THRESHOLD} < +/+/\text{filteredF}
\]

Putting all the parts together will reveal the complete functional specification (the architecture) of this raster detector:

\[
\text{RasterDetected} =: \text{THRESHOLD} < +/+/0(:,<0,1;0,1;3,3); 4 4[, |*: \text{singleTileF}]
\]

The rejection of pure horizontal and vertical rasters is not given here but is based on the typical power distribution between DC and accumulated AC bands.

**Implementation**

Although the specified functionality is executable it is still far away from the target language (in our case C).

The goal of the implementation phase (see Fig. 2) is to work towards a realisation without compromising or altering the functionality (architecture). In this section only a few implementation problems are addressed.

**Problem analysis**

Implementation of raster detection in software on a standard PC with a processing time of less than 0.5s requires new techniques, such as:
• reduction of dimensionality
• reduction of data size
• use of look-up tables (LUTs) to speed up Fourier transforms
• reduction of spectrum information to only the two most dominant power bands.

The last two are not elaborated here: we will elaborate on the other two techniques:

• **Reduction of dimension**: Current PC processors are not equipped to deal with Fourier transforms of a single 2D tile. To do so the 2D problem must be transformed in real time to simpler multiple 1D problems.

• **Reduction of data size**: A good estimate of frequency content can be made by finding the number of times the grey values within each byte cross the average values of rows and columns (see Fig. 11).

Reduction of data size has to be performed for each tile, and within a tile for each of its rows and each of its columns. The following derivation shows the reduction for rows for a single tile (most upper left tile in the tiled bitmap tiles). This tile, again the upper left one, is given as Fig 12.

The averages per row are computed via a so-called fork construct, a function composition. Averaging rows includes summation over rows (+/%), counting the number of members (#) and taking the quotient of both (%); concisely notated by (+/% % #).

The average of each row is given below by vector `threshRows`:

```
] threshRows =: (+/%1 #) singleTile
152.875 154.625 152.25 153.5 153.5 154 153.375 153.75
```

Comparing each row with this row average (`singleTile > threshRows`) will yield the result as described in Fig. 11 (see boolean matrix, Fig. 13):

```
] fingerprintRows =: #. singleTile > threshRows
82 181 82 184 82 170 90 232
```

**Summary**

The problem size is reduced in several steps until it can be realised on a standard PC processor. Such reduction with acceptable quality degradation is performed in the following steps:

1. 2D to multiple 1D transforms reduction
2. removal of DC power and other non relevant frequency content
3. reduction of grey level information to numbers of zero crossings
4. reduction of generated spectra to only the two highest power bands.

The complexity of this particular detector was sufficiently low that a C implementation could be made by hand with the necessary LUT tables generated by a C source, although J could well have been used for the complete development, implementation as well as design.
Conclusions

Raster detection

The goal of both raster form and orientation-independence is realised (see Fig. 14 to get an impression of the enhancements). The white blobs are clearly much smaller than before.

The global raster detection is realised within 0.13 s on average per A4 page on a 600 MHz Pentium III. No

Methodology and language issues

Using J for this and other applications helped build up experience of how IT professionals use the language. The following statements summarise their findings:

Positive features of J

1. It supports the requirement phase, starting from problem analysis to functional specification (architecture).
2. It supports the design and the recording of design decisions and alternatives during the implementation phase.
3. It enables early executability of specifications, and thereby reduces proliferation of errors further on in the software development process.
4. It enables functionality to be expressed at any given level of abstraction, from a high abstract level down to production code.

Negative features of J

1. The relationships between J concepts and the development methods used in the architectural and design processes are not completely straightforward.
2. More specifically, it is not immediately clear how to use the benefit of executable specifications, that is, by gradually replacing abstract specifications with implementation code.
3. Deriving an architecture and developing an implementation do not in themselves require a specific computing language.
4. J does not resemble other popular languages and has an esoteric image, which makes it difficult to switch between J and other languages.

To conclude, using J for problem analysis and system design is complicated by factors which have nothing to do with the language itself, but which can divert attention from the primary goal of putting a right methodology in place. Education is the way to address these issues, through courses at technical colleges and universities. However, for engineers who are willing to spend time applying this sort of methodology and learning the language J, the effort involved will certainly pay off.

Reference


© IEE 2002

The author is with Océ Technologies BV, PO Box 101, 5900MA Venlo, Netherlands.