$

Visual Programming for Machine Vision

By

Paul F. Whelan[1]

This chapter will outline the development of a visual programming environment for machine vision applications, namely *JVision*[2] [WHE-97a, WHE-97b]. The purpose of JVision is to provide machine vision developers with access to a non-platform specific software development environment. This requirement was realised through the use of Java, a platform independent programming language. The software development environment provides an intuitive interface which is achieved using a drag and drop, block diagram approach where each image processing operation is represented by a graphical block with inputs and outputs which can be interconnected, edited and deleted as required. Java provides accessibility, hence reducing the workload and increasing the deliverables in terms of cross-platform compatibility and increased user base. JVision is just one example of a such a visual programming development environment for machine vision, other notable examples include Khoros [KRI-99] and WiT [WiT-99]. See [JAW-96, GOS-96] for details on the Java programming language.

$.1 Design Outline

JVision is designed work at two levels. The basic level allows users to design solutions within the visual programming environment using JVisions core set of imaging functions. (Currently JVision contains over 200 image processing and analysis functions, ranging from pixel manipulation to colour image analysis. A

---

[1] *http://www.eeng.dcu.ie/~whelanp/home.html.*

[2] JVision was designed and developed at the Vision Systems Laboratory, Dublin City University, Dublin 9, Ireland. See *http://www.eeng.dcu.ie/~whelanp/jvision/jvision_help.html* for more details, and for information on acquiring this package.

full listing of all the functions available can be found at the JVision web site.) At the more advanced developers level, JVision allows users to integrate their own functionality, i.e. to upgrade through the introduction of new image processing modules.

The following sections outline the key issues involved in the development and use of a visual programming environment for machine vision. While may of the issues discussed are common to a wide range of visual programming environments, this chapter concentrates on the issues specific to a machine vision development system.

$.1.1 Graphical User Interface (GUI)

The graphical user interface consists mainly of a canvas where the processing blocks reside. The processing blocks represent the functionality available to the user. Support is provided for handling positioning of blocks around the canvas or workspace, the creation and registration of interconnections between blocks and support for double clicking. See Figure $.1 for an example of the JVision GUI. The lines connecting each block, represent the path of the image through the system, with information flowing from left to right. Some of the blocks can generate child windows, which can be used for viewing outputs, setting parameters and selecting areas of interest from an image. If each block is thought of as a function then the application can be thought of as a visual programming language, the inputs to each block are similar to the arguments of a function, and the outputs from a block are similar to the return values. The advantage in this case is that a block can return more than one value. The image processing system can be compiled and executed as with a conventional programming language with errors and warnings being generated depending on the situation. Warnings are generally associated with the failure to connect blocks correctly.

$.1.2 Object Oriented Programming

The Object Oriented Programming (OOP) paradigm allows us to organise software as a collection of *Objects* that consist of both data structure and behaviour. This is in contrast to conventional programming practice that only loosely connects data and behaviour. The Object Oriented approach generally supports five main aspects: *Classes, Objects, Encapsulation, Polymorphism* and *Inheritance*. Object Orientation (OO) encourages modularization (decomposition of the application into modules) and the resuse of software, in the creation of software from a combination of existing and new modules.

Classes allow us a way to represent complex structures within a programming language. Classes have two components. *States* (or data) are the values that the object has and *methods* (or behaviour) are the ways in which the object can interact with its data, i.e. the actions.

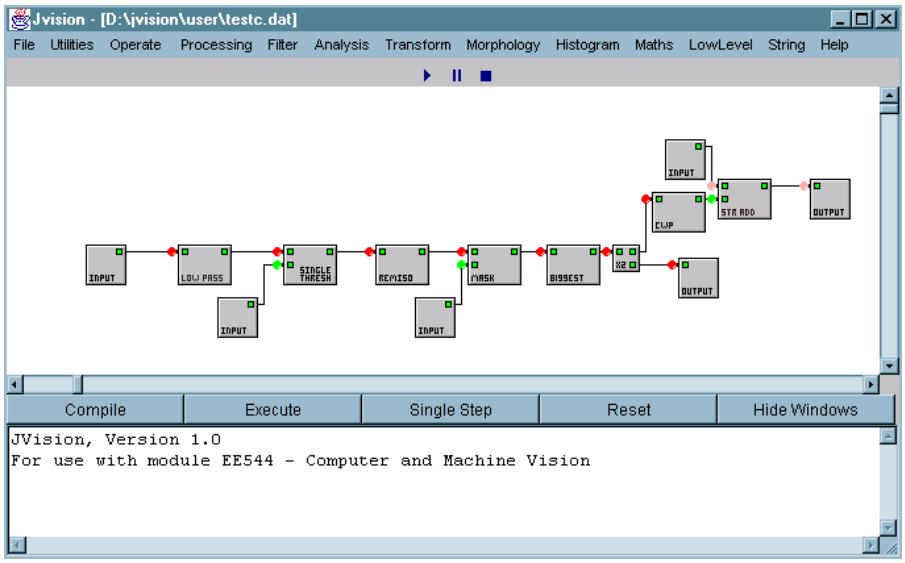


Figure $.1, A typical JVision canvas.

Objects are instantiated, in that they exist in the computers memory, with memory space for the data. Instantiation is the creation of an object from the class description. An object is a discrete, distinguishable entity. For example, *my car* is an object, whereas the common properties that exist with *my car* and *every other car* may be grouped and represented as a class called *Car*. Objects can be concrete (a car, a file on a computer) or conceptual (a database structure) each with its own individual identity.

Encapsulation provides one of the major differences between conventional structured programming and object oriented programming, it enables methods and data members to be concealed within an object in effect making them inaccessible from outside of the object. In effect a module must hide as much of its internals as possible from other modules. Encapsulation can be used control access to member data forcing its retrieval or modification through the objects interface which should incorporate some level of error checking. In strict object oriented design an objects data members should always be private to the object, other parts of the program should never have access to that data.

A derived class inherits its functions or methods from the base class, often including the associated code. It may be necessary to redefine an inherited method specifically for one of the derived classes, i.e. alter the implementation. So, *polymorphism* is the term used to describe the situation where the same method name is sent to different objects and each object may respond differently. Polymorphism has advantages in that it simplifies the Application Programming Interface (API) [SUN-99] and also a better level of abstraction can be achieved.

Object Oriented languages allow a new class to be created by extending some other class, so *inheritance* enables the creation of a class which is similar to one previously defined. An inherited class can expand on, or change the properties of

the class from which it is inherited. In traditional programming, modification of existing code would be required to perform a function similar to inheritance introducing bugs into the software as well as generating bulky repetitive source code. The level of inheritance can be controlled by the use of the key words *public, private* and *protected.*

$.1.3 Java Image Handling

Java provides an *Image* class for the purpose of storing images. This class contains useful methods for obtaining information about the image. The most useful of these methods are described in Table $.1. The image object can be displayed on the canvas using the *drawImage()* method which is a member of the *Graphics* class, this method allows the scaling of an image using a bounding box. The scaling function is particularly useful in performing basic magnification tasks.

Method	Description
getWidth(ImageObserver)	Returns the width of the image object, if the width is not yet known the return value is -1.
getHeight(ImageObserver)	Returns the height of the image object, if the height is not yet known the return value is -1.
getSource()	Returns the image producer which produces the pixels for the image. An image producer is sometimes used in the generation of a filtered image.
ImageObserver	An image observer is an object interested in receiving asynchronous notifications about the image as the image is being constructed.

Table $.1 Methods of the image class.

An image consists of information in the form of pixel data, in the case of a Java image object, a pixel is represented by a 32 bit integer. This integer consists of four bytes containing an alpha or transparency value, a red intensity value, a green intensity value and a blue intensity value. This representation of pixel data is referred to as the default RGB colour model. The structure of a pixel using this colour model is given in Figure $.2. For the purposes of JVision the alpha or transparency value is always set to its maximum value as all images are required to be fully opaque for display purposes, hence the alpha value can be ignored when processing an image. Also for grey-scale images the values for each of the colour planes will be the same hence only the value for one of the colour planes need be extracted in order to obtain the grey intensity value for that pixel, this approach increases the speed at which an image can be processed.

$.1.4 Image Processing Strategy

The key requirement of the image processing and analysis strategy is generation of a robust method for image flow control. This is achieved through the introduction of the sequential block processing algorithm or the block manager. Blocks appear as boxes with a user definable number of inputs and outputs depending on which image-processing function is being implemented. The only information the sequential block-processing algorithm requires is the actual state of each of the blocks in the image processing system. Blocks request attention by setting an internal state variable, which is polled by the block manager, rather than generating an interrupt. A block can have one of several states and the block manager will act according to the state of the block. The complete list of currently available block states and their description are given in Table $.2.

The purpose of the sequential block processing algorithm is to bring all the processing blocks in a system to their steady state. This is achieved by monitoring for two main states, the operation of the algorithm can be summarised as follows. Any block which receives new information on any of its inputs eventually signals that it is WAITING_TO_PROCESS. When the algorithm sees a block signalling WAITING_TO_PROCESS it calls the *processImage()* method of the relevant block thus causing the block to signal WAITING_TO_SEND. When the algorithm sees a block signalling WAITING_TO_SEND it calls the *sendImage()* method of the block in question which results in the block returning to its STEADY_STATE. The algorithm loops through all the blocks in the linked list until they all signal STEADY_STATE when this occurs the algorithm terminates.

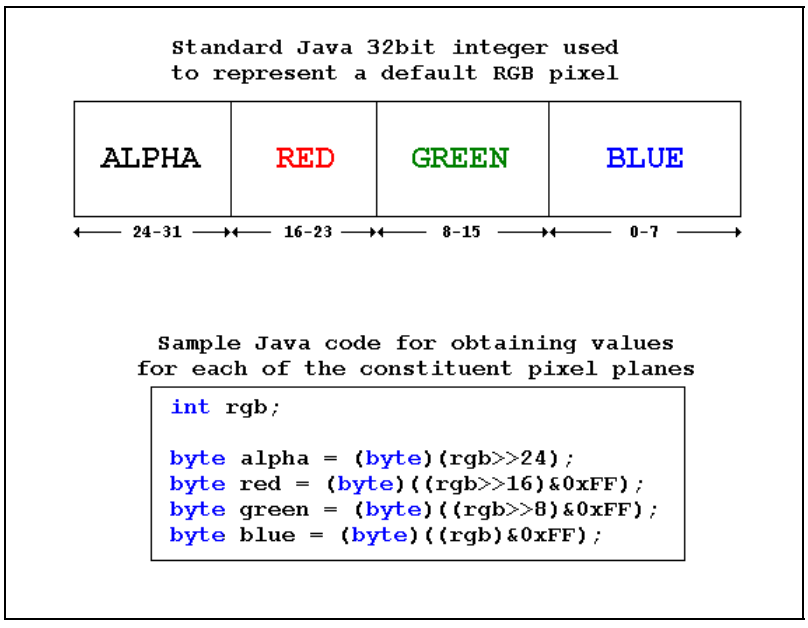


Figure $.2, The default RGB colour model.

Name of State	Description
STEADY_STATE	A block in STEADY_STATE is ignored by the block manger as it is assumed to have processed its image and requires no further attention.
WAITING_TO_PROCESS	A block which is WAITING_TO_PROCESS has received an image from one of its inputs and has verified that there are no images inbound on any other of its inputs. This checking back is required to avoid glitch-like operations evident in digital systems where asynchronous inputs to gates occur.
WAITING_TO_SEND	A block is WAITING_TO_SEND if it has processed an image and sent the image on to its output node.
WAITING_TO_RECEIVE	A block is WAITING_TO_RECEIVE only if it has more than one input. When an image is received on any input a check back function is called to ascertain if there is any more inbound image data. If so wait for that data before going in to the WAITING_TO_PROCESS state. Asynchronous inputs are caused by intensive processing taking place on one input to a block
WAITING_TO_FEEDBACK	The only block which can implement the WAITING_TO_FEEDBACK state is the feedback block. Feedback can only occur when all the other blocks have settled into steady state.

Table $.2 JVision block states.

In order to integrate the type of image processing required certain changes to the GUI must be made, these changes are outlined below.

- Addition of image storage capabilities to the connectors both male and female is required as inter-block communications occur at the connector level thus the input and output images of a block must be stored at this level.
- Addition of *getImage()* and *setImage()* methods to each type of connector is required so that communication of images between male and female connectors is possible, if a block wishes to process an image it reads that image from its input connector and when the processing is complete it writes the new image to its output connector.
- The possible block states as outlined in Table $.2 must be added to the block template so that the so that the sequential block processing algorithm may be

incorporated into the GUI and thus control the flow of image information throughout the system.

- The definition of possible block types must also be added to the block template so it can be determined where an images originates and terminate.
- The addition of the *processImage()* method to the block template is also required, this method is declared as abstract and must be overwritten in any inherited versions of the block template class. The algorithm contained in the *processImage()* method provides the only difference between each of the blocks available with the JVision.

$.2 Data Types

This section summarises the various data types currently supported by JVision visual programming environment.

$.2.1 Image

Images are the best catered for data types in the JVision library. Dark green nodes correspond to colour images and red nodes correspond to grey scale images. If a colour image is specified for a grey scale input then the grey scale representation of the image is obtained and used in the consequent processing. Note if the image viewer frame does not re-render properly after resizing a refresh can be forced by clicking on the image.

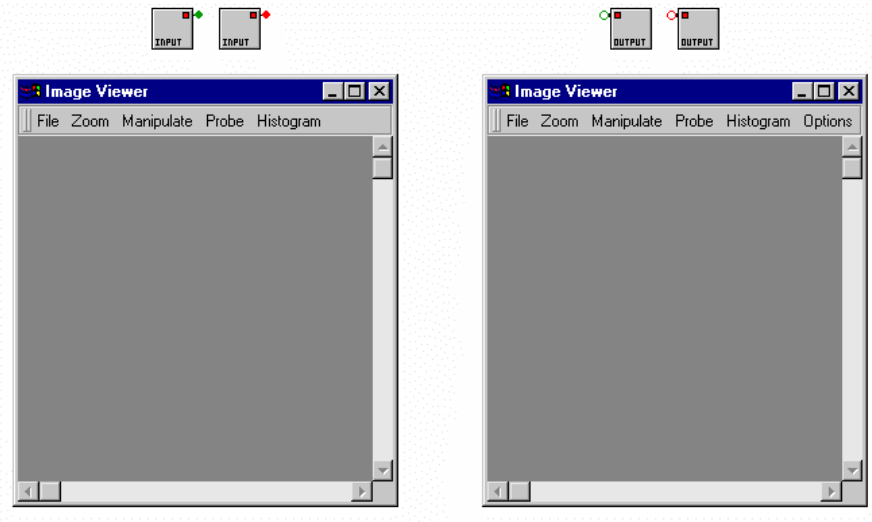


Figure $.3, Image data types.

$.2.2 Integer

Integer values may be used in conjunction with maths blocks or other image processing blocks. The range of the Java integer is between -2,147,483,648 and 2,147,483,647. These are indicated by light green nodes.

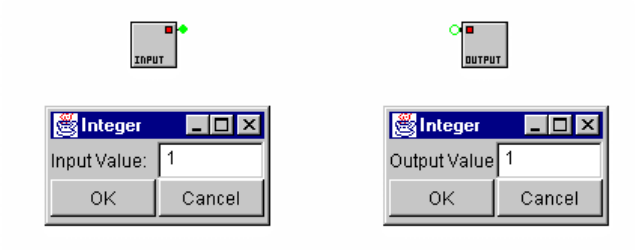


Figure $.4, Integer data types.

$.2.3 Double

Double values may also be used in conjunction with maths blocks or other image processing blocks. The range of the Java double is between -.7976931386232e308 and 1.79769313486232e308. These are indicated by dark blue nodes.

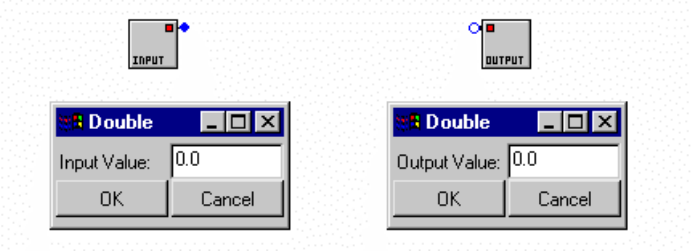


Figure $.5, Double data types.

$.2.4 Boolean

Boolean values are used in the interface between components of the maths library and the conditional processing blocks. The value of a Boolean variable may be either TRUE or FALSE. These are indicated by orange nodes.

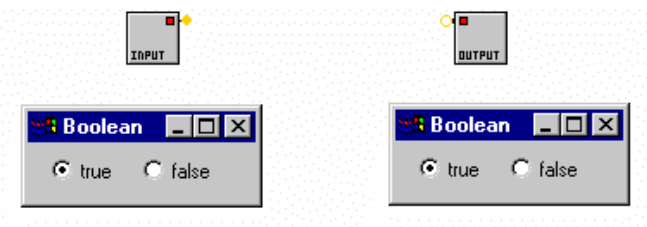


Figure $.6, Boolean data types.

$.2.5 String

Strings values may be used to alert the user to certain results. Any of the above variables integer, double, Boolean may be appended to a string using the string add block. Note that for file saving purposes a string may not contain a new line. These are indicated by pink nodes.

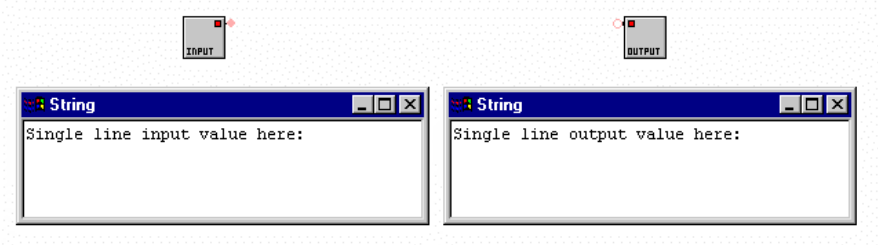


Figure $.7, String data types.

$.2.6 Integer Arrays

Integer arrays are used to provide the mask input for the convolution filter or the structuring element for morphological operations. In the case of the morphological blocks an 'x' in the structuring element corresponds to a don't care statement (in fact an non-valid integer has the same effect, this includes spaces). These are indicated by light green nodes.

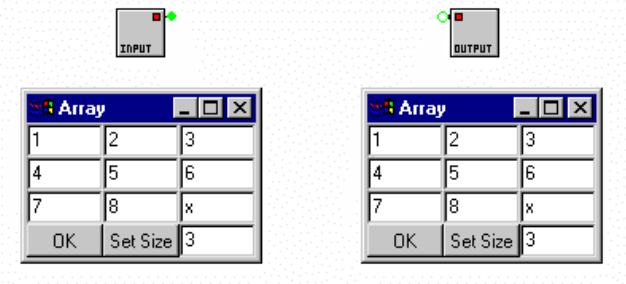


Figure $.8, Integer arrays.

$.3 Non-linear - Feedback Blocks

The non-linear blocks supplied with JVision can be used to conditionally and repeatedly process any type of variable from images to strings. A feedback loop implemented in any of the instances of the blocks described below must contain some kind of operation otherwise they serve no purpose and an error will occur.

$.3.1 Feedback

Feedback is an extremely useful mechanism for implementing a repetitive set of operations. In the example outlined in Figure $.9 multiple dilations are applied to the input image (the top most input on the left side of the feedback block). The number of times the dilation operation shall be applied is specified by the integer input block (this corresponds to the middle input in the feedback block). Once the image has been processed the specified number of times it may be further processed. The result from the feedback loop is obtained from the top most output on the right side of the feedback block. (Figure $.9).

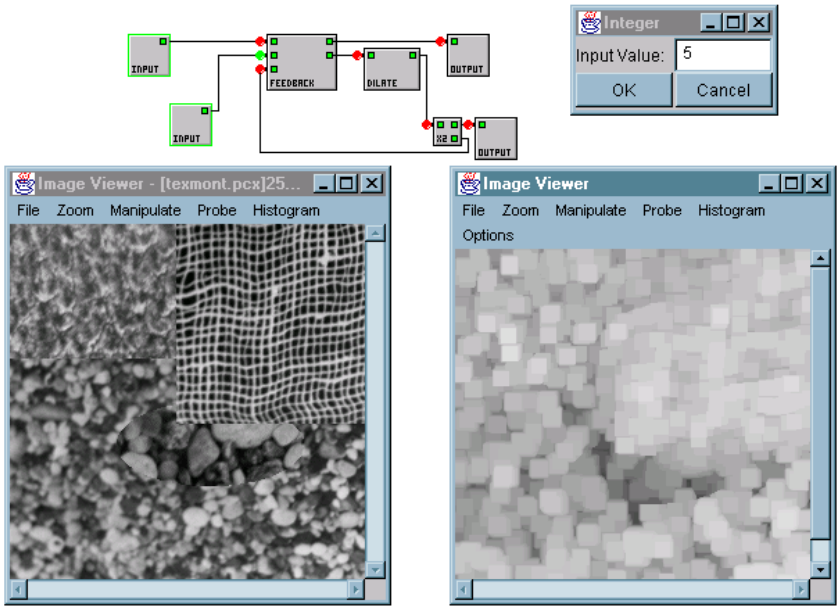


Figure $.9, Feedback example. Multiple dilations of a grey scale image.

$.3.2 FOR Loop

Implementation of the FOR loop is similar to that described previously for the feedback structure, except in this case a loop variable is available to the feedback loop for further processing. Also the single integer input from before is replaced with three inputs which represent the start value, finish value and the loop increment. In fact the implementation here is equivalent to the following C code:

```
For(x=start;x<finish;x+=increment)
```

where start is the top most integer value, finish is the centre integer value and increment is the lower integer value. (Figure $.10).

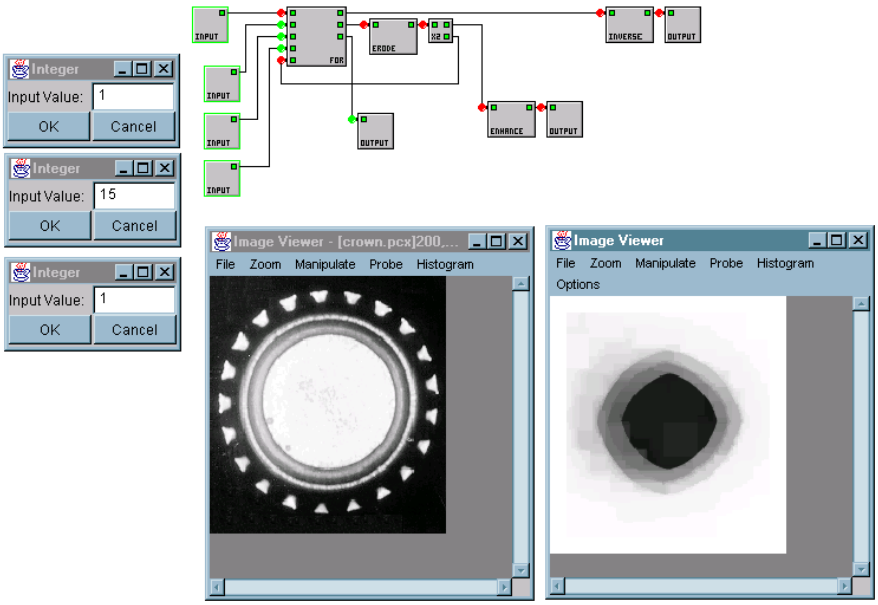


Figure $.10, FOR loop example. Multiple erosion of a grey scale image.

$.3.3 IF ELSE

The IF ELSE structure is yet another implementation of the feedback mechanism. In this case the feedback operation is applied only once, if at all. The decision whether processing of the data should be performed using the IF or the ELSE path is based on the status Boolean input variable. In the case of the JVision canvas illustrated in Figure $.11, if the Boolean input is TRUE then the *Smallest Intensity Neighbour (SIN)* filter is applied to the image. If the Boolean input is FALSE then the *Largest Intensity Neighbour (LIN)* is applied.

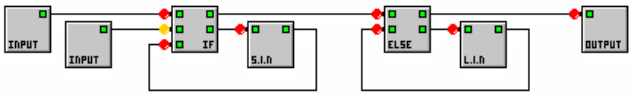


Figure $.11, IF ELSE example.

$.3.4 Nesting

Another key feature of JVision is that it allows multiple nesting of feedback structures. For example the operation implemented in Figure $.12 is low level image negation which performs a pixel by pixel raster scan of the image. The two FOR structures which represent the vertical index and the horizontal index

respectively. The vertical index block ranges from 0 to the image height in steps of 1, i.e. one pixel at a time.

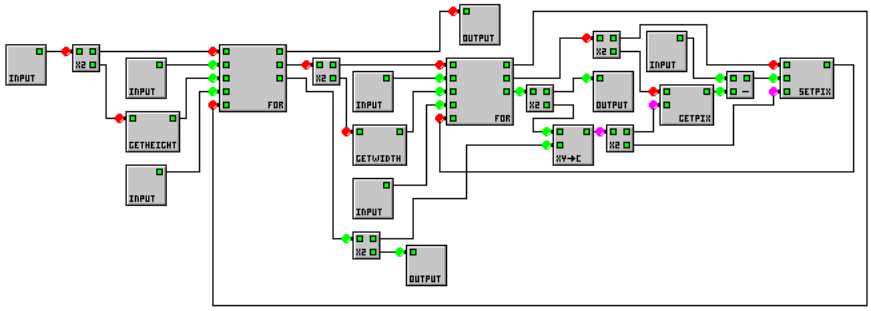


Figure $.12, Nesting. Pixel by pixel negation.

$.4 Visual Programming Environment

Blocks are selected from the menu system as required, then placed in the workspace. Interconnections are made between blocks following a strict 45 degree snap to grid rule system. When the processing system is set up, input images must be specified. This is done by double clicking on the relevant input block this causes an image viewer frame to appear. The *open* option is then selected from the file menu and the appropriate file name is chosen by navigating the file dialog box directory structure. If the file selected contains raw image data then the dimensions of the image must be specified before the image can be loaded. There is currently support for 9 input graphics file formats, see Section $.4.1. Images can be saved in either Microsoft Windows Bitmap (BMP) or PC Paintbrush (PCX) formats.

Initialisation of certain block parameter may be required throughout the system. If they are not set the parameters will assume to their default values. A system may be compiled using the compile button if successful then the system can be executed. System parameters can be adjusted and the system may be reset and executed again until the desired response is obtained. At any stage blocks may be added or removed from the system. A block can be deleted by clicking on the relevant block in order to highlight it, then pressing the delete key. If the delete key is not available which may be the case with certain keyboards then the ‘d’ key will perform this same operation.

Once a programme is executed each block is highlighted as it is processed. This enables novice users to get a sense of the relative speeds of the various processing options available within the JVision environment.

$.4.1 Interpretation of Graphics Files

The first problem which needs to be addressed in the development of any image processing software is interpretation of image resource files containing bitmap or raster information. Java provides support for the two most common file formats

found on the Internet, Graphics Interchange Format (GIF) and Joint Photographic Experts Group File Interchange Format (JPEG). Unfortunately these file formats are not used for computer vision applications as the compression techniques which they employ distort image information. The pixel error introduced by either compression technique is typically 1%, although this may not be visually apparent but it does result in unacceptable information loss.

In order to preserve image information, support for several non-corrupting file formats has to be implemented in Java, the three main formats required are raw image data (BYT), PC Paintbrush (PCX) and Tagged Image File Format (TIFF), see below for the definitive list of file formats supported by the JVision [MUR-96].

- *BMP*: Microsoft Windows Bitmap (BMP), a bitmap image file using Run Length Encoding (RLE), supports a maximum pixel depth of 32 bits. A maximum image size of 32K x 32K pixels.
- *BYT*: Raw image data, grey-scale only with a maximum pixel depth of 8 bits the data is stored with the first byte of the file corresponding to the top left hand corner of the image. No header is contained within the image and details about the dimensions must be supplied by the user. This file format preserves image data, no information lost since no compression is used.
- *GIF*: Graphics Interchange Format (GIF) is a bitmap file which utilises Lemple-Zev-Welch (LZW) compression, is limited to a maximum colour palette of 256 entries and a maximum image size of 64K x 64K pixels.
- *JPEG*: Joint Photographic Experts Group (JPEG) File interchange format is a bitmap file utilising JPEG compression, an encoding scheme based on the discrete cosine transform. It has a maximum colour depth of 16.7 million colours and a maximum image size of 64K x 64K pixels.
- *PCX*: PC Paintbrush (PCX), a bitmap file using either no compression or RLE, allowing image data to be kept intact, A maximum colour depth of 24 bits is available and a maximum image size of 64K x 64K pixels.
- *PGM*: Portable Greymap Utilities (PGM), a bitmap file uses no compression hence allowing image data to be left intact. It has a maximum grey-scale depth of 256.
- *RAS*: Sun Raster Image (RAS), a bitmap file format using either no compression or Run Length Encoding (RLE) supports a maximum of 16.7 million colours and has an undefined maximum image size.
- *RAW*: Raw image data, this has a similar specification to the BYT format described except that colour image data also supported. The image data is stored in RGB triplets with the triplet representing the upper left hand corner of the image.
- *TIFF*: Tagged Image File Format (TIF) a bitmap file using a wide range of compression techniques, uncompressed, RLE, LZW, International Telegraph and Telephone Consultative Committee (CCITT) Group3 & Group4 and JPEG supports a maximum pixel depth of 24 bits (16.7 million colours) and a maximum image size 4 G pixels.

$.4.2 Plug in and Play Architecture

The main objective here is to allow the user to write code for a new block and integrate it into the JVision without the need to recompile the source code. To do this a dynamic class loader must be written to allow a class, in this case an inherited version of the main *Block* class, to be imported into the application after it has been compiled. The name of the class referred to above or more precisely the list names of block classes which together create the JVision must be made known to the application, so it may import them as required. This is implemented in a similar manner to the method in which the resource file information is made available to the file dialog box i.e. through the use of a scripting language. In this case the file called *menubar.rc* contains the list of processing blocks available to the user, the block names are categorised into functional groups e.g. filters, transforms etc. A limitation associated with this approach is that the name of the block must be an exact match for the name of the class which implements the functionality of that block. Although this is easily overcome as the name of the class usually gives a good description of the function which it performs. The scripting language uses the four tags outlined in Table $.3 in the generation of the main JVision menu bar.

Tag	Meaning
<number of menus>	The value directly after this tag specifies the number of menus described by the file. This number is used to initialise an array of Java menu objects and must be an integer.
<menu>	The new line terminated string following this tag is used as the menu name. The strings following the first line specify the menu items i.e. the individual blocks.
</menu>	This tag marks the end of a menu, the file interpreter now waits for another menu or the end of file tag.
<end>	This tag informs the file interpreter that all the menus and menu items have been updated and to close the input stream.

Table $.3 Tags for the automatic generation of the menu bar.

When the JVision application is initialised it reads the data from the *menubar.rc* file with this data it constructs the menu bar for the main window automatically. Each menu in the menu bar corresponds to a menu block in the *menubar.rc* file, the name of the menu being specified by the first string in the menu block and the menu contents being specified by the remaining strings in that menu blocks. This

method of setting up the menu bar performs two tasks. It means the user does not need to recompile the code to update the menu bar and hence the inventory of processing blocks. It also means that when a menu event is handled, the argument specifying the event is the name of the new block to be added to the workspace. This block name is used in conjunction with the class loader to load the selected block, dynamically import it and then add it to the linked list and eventually to the workspace. The method by which the menu bar is automatically updated is outlined in Figure $.13. Note that the dashed lines in the script file correspond to menu separators in the actual menu.

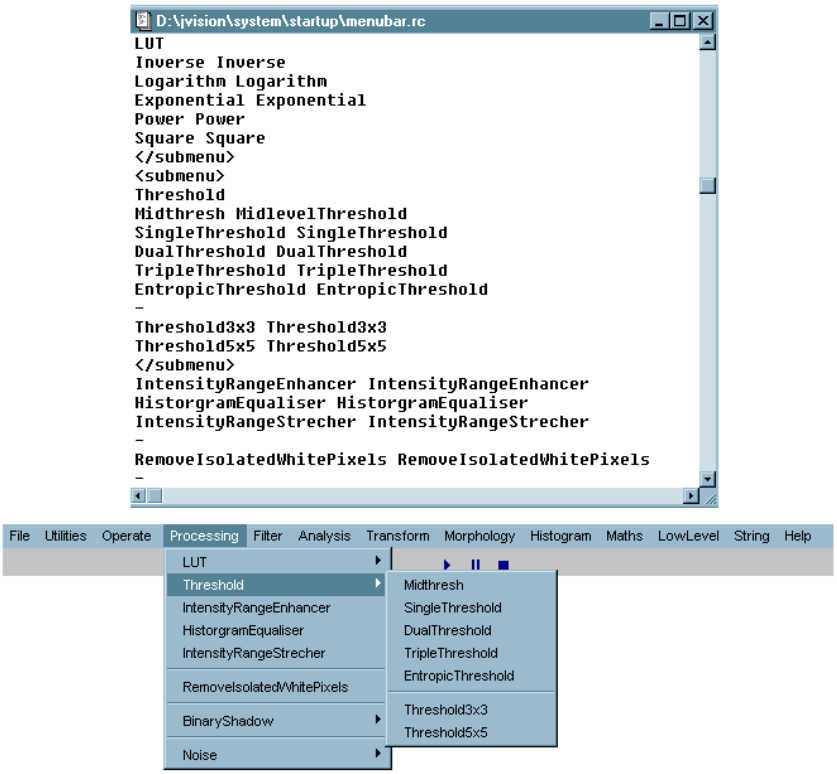


Figure $.13, Automatic menu bar generation.

$.5 Image Viewer and Tools

JVision provides several image investigation tools, horizontal and vertical intensity scans, normal and cumulative histograms, pseudo colour tables and a 3D profile viewer. These functions are common to both the colour and grey scale image viewer frames.

$.5.1 Horizontal and vertical scans

The horizontal and vertical scans tools provide cross-sectional intensity maps of the average intensity of the displayed image, this means that it works with both grey scale and colour images. The profile lines may be removed by closing the Horizontal and Vertical Scan windows. Both scans can be applied simultaneously giving the cross hair display illustrated in Figure $.14.

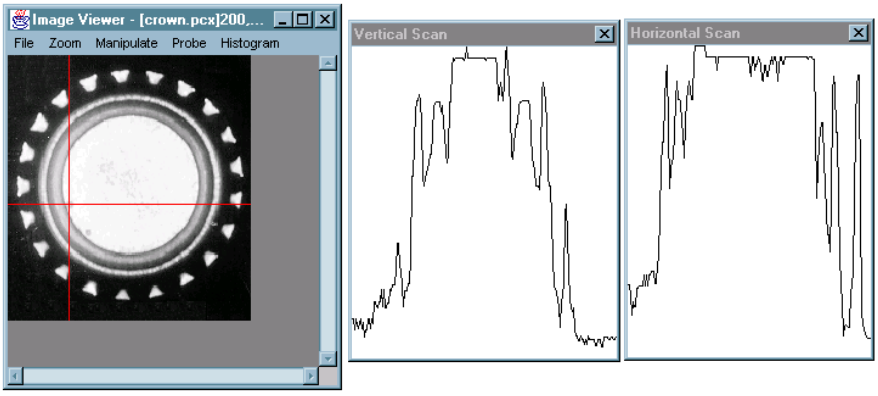


Figure $.14, Horizontal and Vertical Scans.

$.5.2 Histograms

The displayed image may be represented using the histogram function in either normal or cumulative mode.

$.5.3 Pseudo Colour Tables

JVision provides a selection of pseudo colour tables which may be used to re-render an image. This is especially useful when it is required to distinguish between several grey levels of similar intensity. A random pseudo colour table is also provided. This has been found to be useful when dealing with images which have been processed using the label operation.

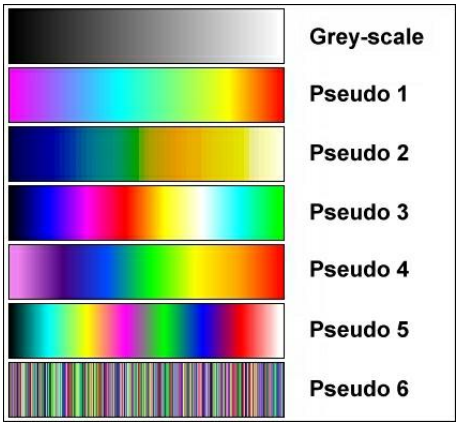


Figure $.15, Grey scale representation of the Pseudo colour tables.

$.5.4 3D - profile viewer.

The 3D profile viewer can be launched only from the image viewer frame when a valid Region of Interest (ROI) is selected. A ROI can only be selected if neither of the other two probing utilities (horizontal and vertical scan) are in operation, as they have precedence over ROI selection. Once the 3D viewer is launched then the profile may be manipulated using the menu system or the 3D navigator which can be launched using the navigate menu. The 3D profile viewer may be applied to a colour image in which case only the intensity of the image is represented.

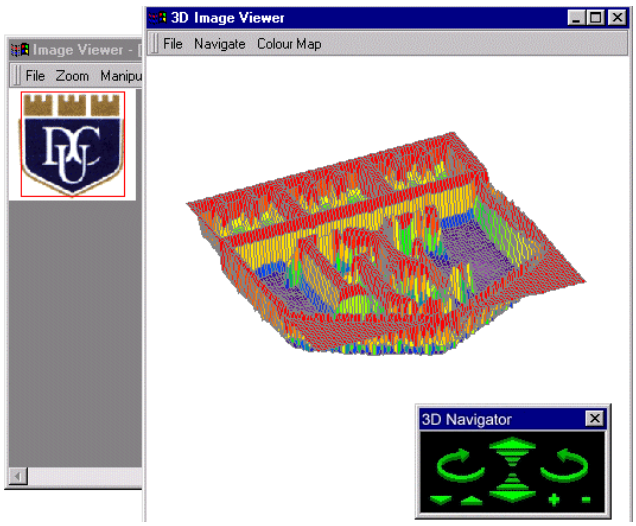


Figure $.16, 3D image viewer. The 3D navigator is shown on the bottom right of this diagram.

In addition to the default grey scale colour table used with the profile viewer we may also use any of the pseudo colour tables described previously in order to distinguish between differing grey levels as well as adding extra depth information to the image.

In addition to using the 3D navigator or the menu bar we may manipulate the profile using the keyboard, as described below. Note that the profile must be brought into focus by clicking on it before any keystrokes are valid.

- UP ARROW - tilt backwards
- DOWN ARROW - tilt forwards
- RIGHT ARROW - spin anticlockwise
- LEFT ARROW - spin clockwise
- MULTIPLY - intensity multiply

- DIVIDE - intensity divide
- PLUS - zoom in
- MINUS - zoom out

$.6 Sample Problems

JVision provides an image analysis software development environment that can work at several levels. For example at a relatively low level we can manipulate individual pixels (Section $.6.1). Alternatively we can use JVisions built in functionality to generated solutions to complex machine vision tasks (Section $.6.5).

$.6.1 Low-level programming.

The solution outlined in Figure $.17 illustrates this for the case where pixels in a circular arrangement are superimposed on a black image. This solution implements the following equivalent code.

```
int  xcent = 128;
int  ycent = 128;
int  radius = 20;
int  colour = 255;

for (int i=0;i<360;i++)
      {
      int x = radius*sin(i);
      int y = radius*cos(i);
      input.setxy(xcent+x,ycent+y,colour);
      }
```

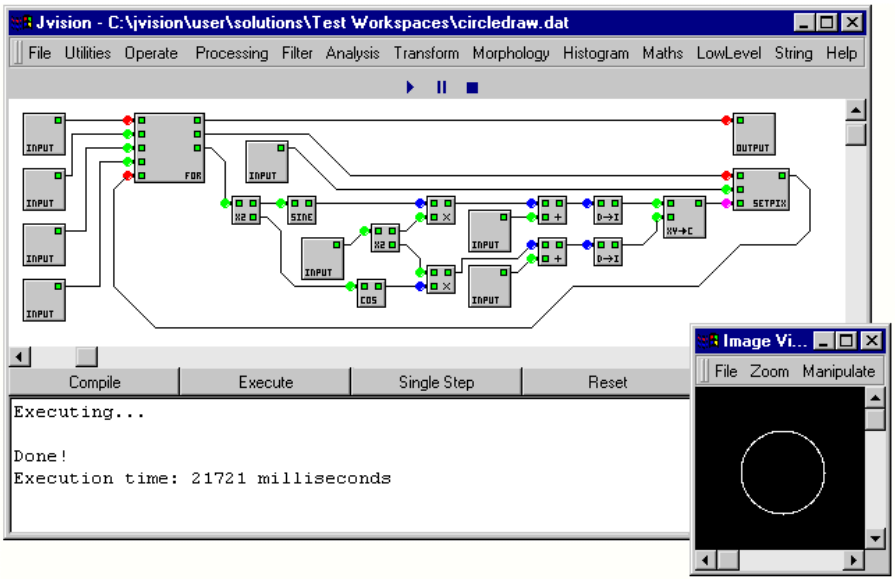


Figure $.17, Low level functionality of JVision.

$.6.2 High-level programming.

Working at a higher level maximises the use of the predefined image processing algorithms. For example the label by location operator is used to implement a naive blob fill algorithm.

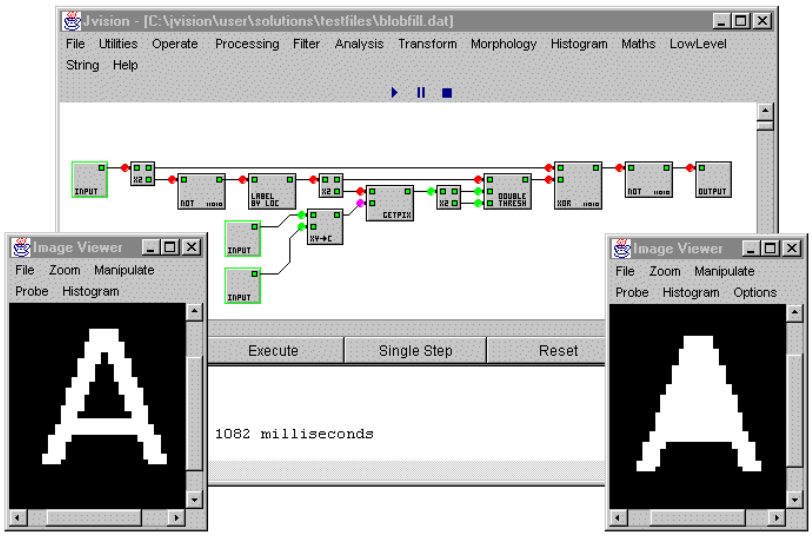
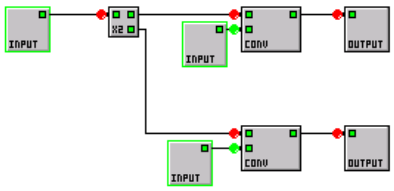


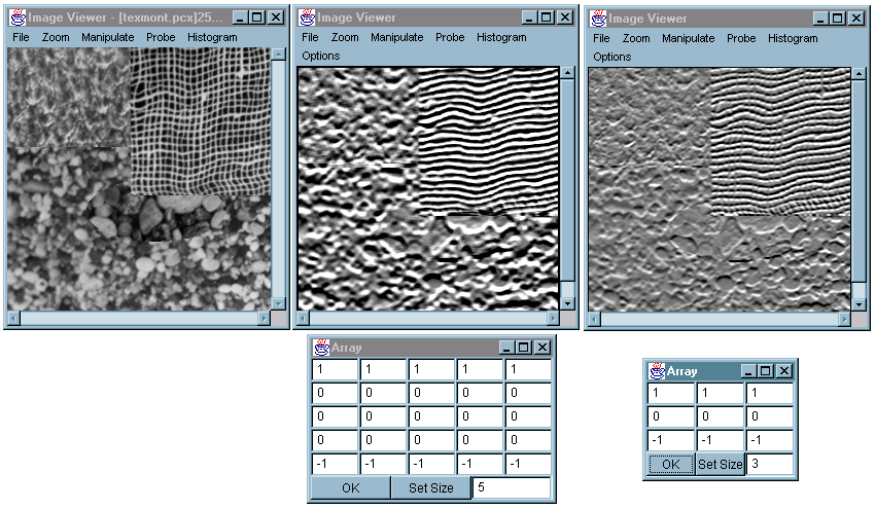
Figure $.18, Blob fill algorithm implementation.

$.6.3 Convolution.

Figure $.19 illustrates the operation of user defined convolution masks of varying sizes (3x3 and 5x5) in extracting horizontal and vertical information from an image.



(a)



(b)

Figure $.19, Convolution. (a) JVision canvas. (b) Results from the 5x5 and 3x3 convolution operators.

$.6.4 Fourier Transform.

At the highest level complex abstract concepts such as Fourier analysis can be implemented, for example the band pass filter illustrated in Figure $.20.

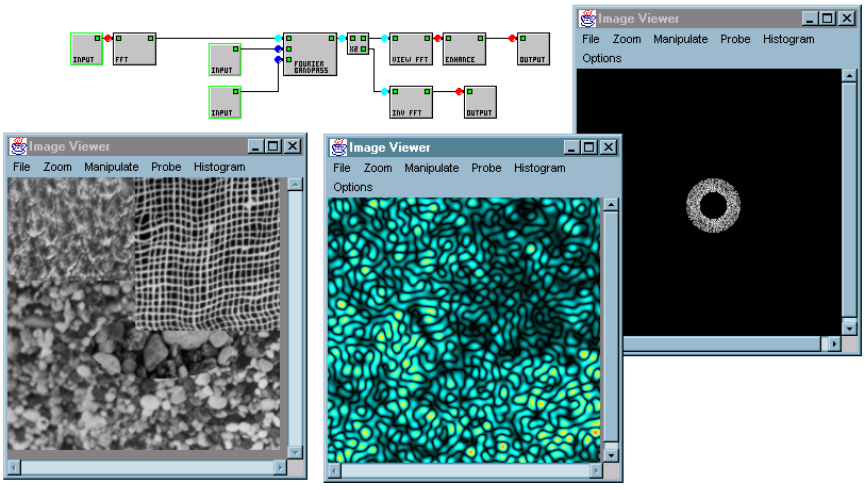


Figure $.20, Fourier analysis: Band pass filter. The original input image is indicted on the lower left. The resultant filtered image is illustrated on the lower right.

$.6.5 Isolate the largest item in the field of view.

The aim of this programme is to find and isolate the largest white region in the scene and indicate the area (in pixels) of this region in an embedded text message. The input image is in PCX format. This is loaded by double clicking the input image box and selection the *crown.pcx* image. A low pass filter is applied to the image, with the effect of blurring the image. The image is then thresholded at grey scale 200. All pixel values below 200 will go to black, the rest will be assigned to white, producing a binary image. You can experiment with the threshold by double clicking the single threshold box and varying the slider bar. Any single isolated white pixels are then removed and a 3 pixel wide black border is drawn around the image. This is implemented to eliminate any white pixels touching the boundary (a necessary requirement for the biggest blob operator). See Figure $.21.

The largest white region is then isolated, and its area calculated by counting the number of white pixels. This is assigned to an integer variable which is then combined with the predetermined text string "The area in pixels is: " and displayed to the user.

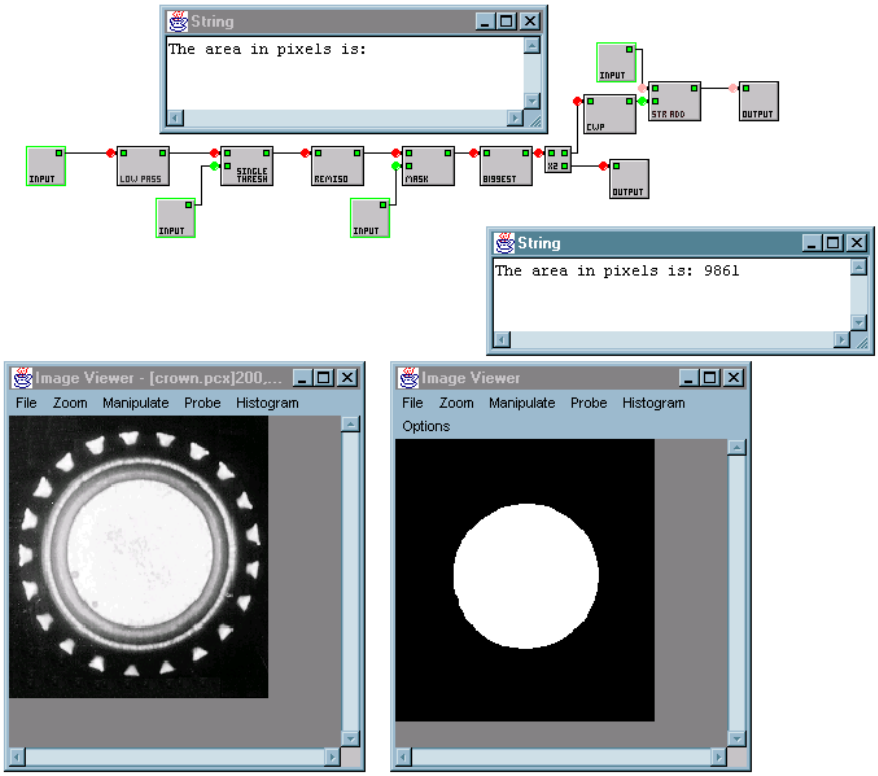
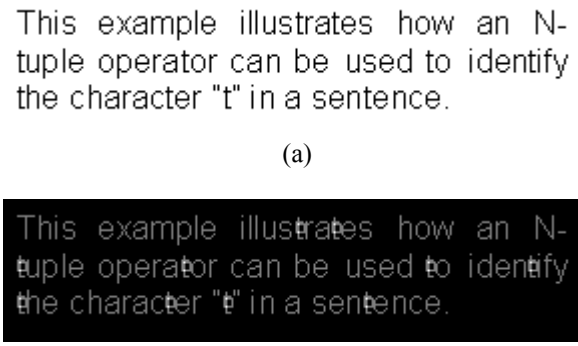


Figure $.21, Find and isolate the largest white region in the scene.

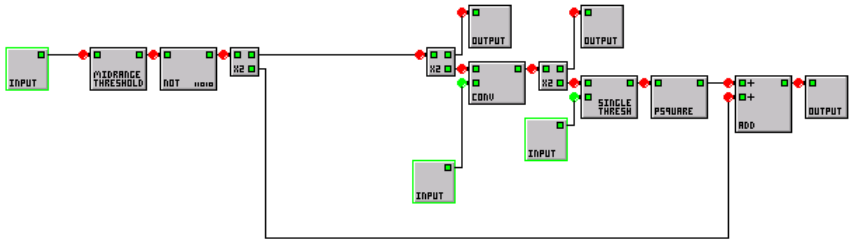
$.6.6 Character detection using the N-tuple operator.

Figure $.22 illustrates how the N-tuple operator can be used to highlight the Arial font character "t" in a sentence for a given orientation. The original image is threshold at the mid grey level and inverted. This image is then convolved with n N-tuple representing the Arial font character "t", resulting in a output image in which the location of the desired character is highlighted by a white point. The threshold operator is the applied to isolate these points and the detected characters are emphasised by overlaying them with a small square box. The JVision canvas for this is illustrated in Figure $.22c.



(a)

(b)



(c)

Figure $.22, (a) Original image. (b) Detection of the Arial font "t" character using an N-tuple operator. (c) The associated JVision canvas.

Bibliography

[GOS-96] J Gosling, B Joy, G Steele, *The Java Language Specification (Java Series)*, Addison-Wesley Pub Co. 1996, ISBN: 0201634511.

[JAW-96] J Jaworski, *Java Developers Guide*, Sams Net, 1996. ISBN: 1-57521-069-X.

[KRI-99] Khoral Research, Inc. - http://www.khoral.com/core.html

[MUR-96] JD Murray & W Van Ryper, *Encyclopaedia of Graphics File Formats*, O'Reilly, 1996.

[SUN-99] Java™ Technologies from Sun Microsystems: Products and APIs. http://java.sun.com/products/index.html

[WHE-97a] PF Whelan, Remote Access to Continuing Engineering Education RACeE, *IEE Engineering Science and Education Journal*, Oct 1997, pp205-211, 1997.

[WHE-97b] PF Whelan, *EE544 - Computer and Machine Vision – Online Course*, http://www.eeng.dcu.ie/~whelanp/vsg/outline.html, 1997.

[WiT-99] Logical Vision - http://www.logicalvision.com/default.htm, 1999.