A High-Level Programming Language For Digital Morphology

J.R. Parker
Laboratory for Computer Vision
Department of Computer Science
University of Calgary
Calgary, Albert, Canada T2N-1N4

Abstract

A system for digital morphology is presented in which morphological operators are implemented as basic operation in a programming language, and where basic data types include pixels and images. The resulting language can be used effectively for prototyping morphological software and for teaching the principles.

1. Introduction

Morphology is still considered to be a bit of a 'black art' by many. The effect of a particular structuring element on an arbitrary image can be obscure, and the science of constructing structuring elements to have a particular effect is relatively young and incomplete. Many of the tools needed for experimenting with morphology are really just libraries of procedures and functions, callable from a user written computer program. The complexity of correctly interfacing with code written by another is well known, and the process leads to time consuming debugging sessions. There is an overall lack of confidence in the displayed results in many instances.

To avoid these problems, and to encourage experimentation with morphological techniques, a programming language named **MAX** (Morphology And eXperimentation) has been devised. **MAX** is a very simple language, in the style of Pascal and Modula, the sole purpose of which is to evaluate morphological expressions. **MAX** 'compiles' into C code, which is then linked with a library of morphological functions. A MAX program will call the library routines correctly, vastly simplifying the user interface. There are also flow of control statements and input/output facilities which, again, simplify the demands on the user and lead to greater confidence in the results.

To help explain the use of **MAX** here is a simple program that reads in two images and copies them to new image files:

```
// Test of input and output in MAX.
//
        image a, b;
        begin
// Read in two images
        do a << "a";
        do b << "b";
// Copy them to new files.
        do b>>"copyb";
        do a>>"copya";
end;
```

The '//' characters begin a comment, which extends to the end of the line. The first statements in this program are declarations; all variables must be declared between the beginning of the file and the first begin statement. **MAX** recognizes only three types: IMAGE, which is used for both data and structuring elements interchangeably; INT, which is a traditional integer type; and PIXEL, which is a pair of integers that represent the row and column indices of a pixel in an image. Variables may be declared to be of any of these three types by stating the type name followed by a list of variables having that type. STRING constants are allowed in some cases, but there are no string variables.

The executable part of a **MAX** program is a sequence of statements enclosed by a begin at the beginning and an end at the end. A semicolon (;) separates each statement from the next in a statement sequence, except before an end statement. In the program above we see only one kind of statement: a DO. This is simply the word do fol-

lowed by any legal expression, and permits the expression to be evaluated without assigning the value to anything.

The only operators seen above are '<<' (input) and '>>' (output), in this case applied to images. the expression **a << "a"** reads an image in PBM file format from the file named "a" into the image variable a; if the string constant is the empty string "" then standard input is used as the input file, and if the string is "$1" then the first command line argument is copied into the string, allowing a program to open a file specified at run time. In the program above, two images are read into variables. These are immediately written out again under different names: the output operator '>>' works in the same way as the input operator, creating a PBM image file from the specified image.

MAX has six different types of statement, designed to allow a great deal of flexibility in what kinds of morphological operations can be easily implemented. In summary, the legal statements are:

if (*expression*) **then** *statement*

If the expression evaluates to a non-zero integer (TRUE) then the statement that follows will be executed. The statement can be a sequence of statements enclosed by begin - end.

if (***expression***) **then** ***statement1*** **else** ***statement2***

As above, but if the expression is 0 (FALSE) then statement2 is executed.

loop ... end

Repeat a sequence of code. When the end is reached, execution resumes from the statement following the loop statement. Statements within the loop must be separated by semicolons.

exit ***N*** **when** ***expression***

Exit from a loop if the expression evaluates to a non-zero integer. If N is omitted, the exit branches to the statement following the end of the near est enclosing loop. If N=2, then we escape from the nearest 2 nested loops, and so on.

do *expression*

Evaluate the expression. This is mainly useful for input and output.

message *expression*

Print a message to standard output. If the expression is a string constant then that string is printed on the screen; integers and pixels can also be printed, and images will be printed as a two dimensional array of integers as if they were structuring elements.

Assignment

The assignment operator is ':='. The type of the variable on the left of the assignment operator must agree with the expression on the right of it. If the expression is an image the result of the assignment is a copy of that image.

For a small language there is quite an array of operators in **MAX**, and many of these can operate on all three possible data types. Figure 1 contains a convenient summary of all of the legal **MAX** operators. Parentheses can be used in expressions to specify the order of evaluation. There is no precedence implicit in the operators other than that unary operators will be computed before binary ones, and evaluation is otherwise left to right.

As a tool for education and as a test bed for morphological experimentation and testing of structuring elements, **MAX** is unique (if perhaps not perfect). For example, the a program to perform a binary dilation can now be written:

```
// Dilation using ++
   image a, b;
   begin
    do (a << "$1")++(b<<"$2") >> "$3";
   end;
```

Operator	Left	Right	Result	Description
++	image	image	image	Dilate LEFT by RIGHT.
--	image	image	image	Erode LEFT by RIGHT.
<=	image	image	int	Subset: LEFT a subset of RIGHT?
<=	int	int	int	Less than or equal to (int).
>=	image	image	int	Subset: RIGHT a subset of LEFT?
>=	int	int	int	Greater or equal (integer).
>	image	image	int	Proper subset.
>	int	int	int	Greater than
<	image	image	int	Proper subset
<	int	int	int	Less than
<>	image	image	int	Images not the same
<>	int	int	int	Not equal
<>	pixel	pixel	int	Not equal, pixels.
=	image	image	int	Are LEFT, RIGHT equal?
=	int	int	int	Integer equality.
=	pixel	pixel	pixel	Pixel equality.
=	image	int	int	All pixels in the image equal?
-	int	int	int	Integer subtraction.
-	pixel	pixel	pixel	Vector subtraction.
+	image	image	image	Union of LEFT and RIGHT.
+	int	int	int	Integer addition.
+	pixel	pixel	pixel	Vector addition.
+	image	pixel	image	Add a pixel to n image.
+	pixel	image	image	Add a pixel to an image.
*	image	image	image	Intersection of LEFT, RIGHT
*	int	int	int	Integer multiplication.
<<	image	string	image	Read an image from a PBM file named by the string.
<<	int	string	int	Read an integer

FIGURE 1 - Operators available in the MAX language. (Part 1)

Operator	Left	Right	Result	Description
<<	pixel	string	pixel	Read a pixel (2 ints)
>>	image	string	image	Write an image to a PBM file named by the string.
>>	int	string	int	Write an integer
>>	pixel	string	pixel	Write a pixel (2 ints)
->	image	pixel	image	Translate the image by the pixel.
<-	pixel	image	image	Translate the image by the pixel.
@	pixel	image	int	Membership: pixel iN the image?
[]	[int, int]			Pixel generator. Result is the pixel whose ROW is the first int and whose column is the second
.	image.rows			Each image has 4 attributes: # of rows and columns, and the row & column locations of the origin.
.	image.cols			
.	image.origin_x			These can beaccessed by
.	image.origin_y			Imagename.attrname.
.	pixel.row			
.	pixel.col			

Unary operators:

~	Set complement
!	Allocate an image like given one.
#	The (integer) number of isolated pixels in an image.
-	Integer negation.

Image Generator:

{PIXEL1, PIXEL2, "0110101..."} Generates an image, whose size is given by PIXEL1, whose origin is given by PIXEL2, and whose pixels are specified by the string.{PIXEL1, PIXEL2, "0110101..."}

FIGURE 1 - Operators available in the MAX language. (Part 2)

A MAX program for doing a dilation the hard way (I.E. by translating the structuring element to all pixel positions in the image being dilated and accumulating the union of all of these images) should produce the same result. An example of such a program is:

```
// MAX Program to perform a
// dilation the hard way.
int i,j;
image x, y, z;
begin
    i := 0; j := 0;
// Allocate a result image like x
    y := !(x<<"$1");
// Read the structuring element.
    do z<<"$2";
// For all indices i
    loop
     j := 0;
// For all indices j
     loop
// Is pixel i,j in the image?
    if ([i,j] @ x) then
// Translate SE by i,j and OR the
// result (union) with y
    y := y + (z->[i,j]);
    j := j + 1;       // Next j
// j exceeds maximum column?
    exit when j >= x.cols;
    end;
    i := i + 1;       // Next i
    exit when i >= y.rows;
// i exceeds max row?
  end;

  do y>>"$3";// Output the result.
end;
```

2. Example One - The "Hit And Miss" Transform

The hit and miss transform is a morphological operator designed to locate simple shapes within an image. It is based on erosion; this is natural, since the erosion of **A** by **S** consists only of those pixels (locations) where **S** is contained within **A**, or matches the set pixels in a small region of **A**. However, it also includes places where the background pixels in that region do not match those of **S**, and these locations would not normally be thought of as a match.

Matching the foreground pixels in **S** against those in **A** is a "hit", and is accomplished with a simple erosion **A** - **S**. The background pixels in **A** are those found in $\mathbf{A}^c$, and while we could use $\mathbf{S}^c$ as the background for **S** a more flexible approach is to specify the background pixels explicitly in a new structuring element **T**. A "hit" in the background is called a "miss", and is found by $\mathbf{A}^c$- **T**. We want the locations having both a "hit" and a "miss", which are the pixels:

$$A \otimes (S, T) = (A \ominus S) \cap (A^c \ominus T)$$

As an example, let's use this transform to detect upper right corners. Figure 2a shows an image that could be interpreted as being two overlapping squares. A corner will be a right angle consisting of the corner pixel and the ones immediately below and to the left, as shown in Figure 2b. The figure also shows the "hit" portion of the operation (c), the complement of the image (d) and the structuring element used to model the background (e), the "miss" portion (f), and the result of the intersection of the "hit" and the "miss". The set pixels in the result both correspond to corners in the image.

The MAX program that performs a hit and miss transform is:

```
// Hit and miss transform
image a, se1, se2;
begin
 do a << "$1";
 se1 := {[5,5], [2,1],
  "0000000000110000100000000"};
 se2 := {[5,5], [2,1],
  "0000001100001000000000000"};
 a := (a--se1)*(~a -- se2);
 message a;
end;
```

3. Example Two - Identifying Region Boundaries

The pixels on the boundary of an object are those that have at least one neighbor that belongs to the background. Because *any* background neighbor is involved it cannot be known in advance which neighbor to look for, and a single structuring element that would allow an erosion or dilation to detect the boundary can't be constructed. This is in spite of the fact that an erosion by the **simple** structuring element removes exactly these pixels!

On the other hand, this fact can be used to design a morphological boundary detector. The boundary can be stripped away using an erosion and the eroded image can then be subtracted from the original. This should leave only those pixels that were eroded; that is, the boundary. This can be formally written as:

$$Boundary = A - (A \ominus simple)$$

A **MAX** program for this is:

```
// Boundary extraction
image a, b, c;
begin
  do a << "$1";
// SIMPLE structuring element
  b := {[3,3], [1,1], "111111111"};
  c := (a - (a--b));
  message c;
  do c >> "boundary.pbm";
end;
```

Figure 3 shows this method used to extract the boundaries of the "squares" image of Figure 2. A larger

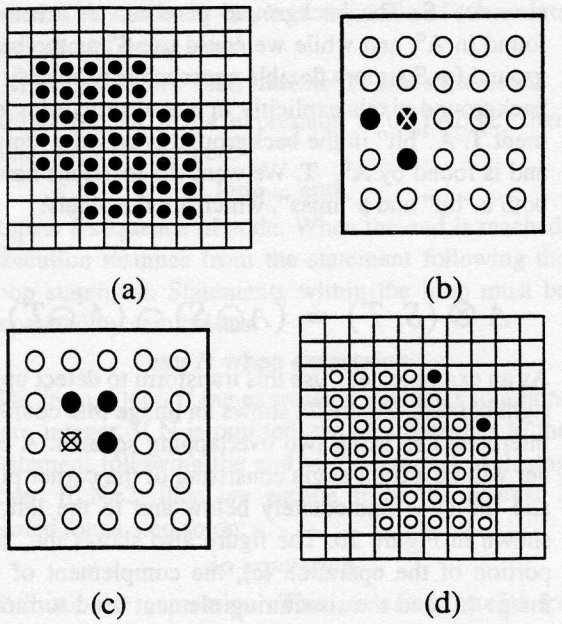

FIGURE 2 - Illustration of the hit and miss transform. (a) The image to be examined. (b) Foreground structuring element for the location of upper right corners. (c) The background structuring element, showing that the three pixels to the upper right of the corner must be background pixels. (d) The result, showing the location of each of the two upper right corners in the original image.

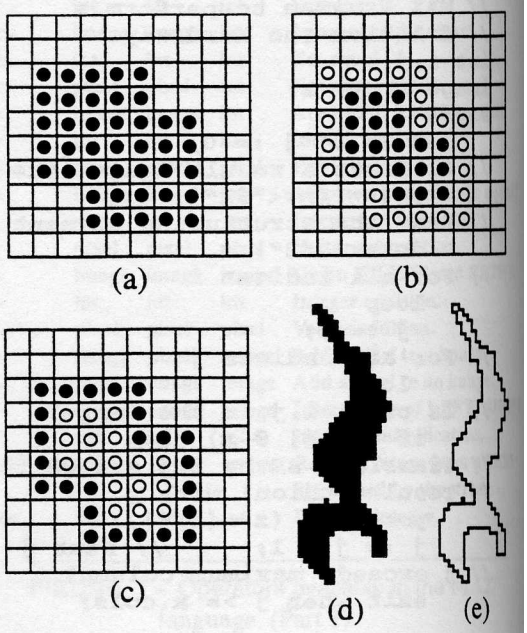

FIGURE 3 - Morphological boundary extraction. (a) The squares image. (b) The squares image after an erosion by the simple structuring element. (c) Difference between the squares image and the eroded image: the boundary. (d) A musical quarter rest, scanned from a document. (e) The boundary of the quarter rest as found by this algorithm.

example, that of a quarter rest scanned from a page of sheet music, also appears in the figure.

4. Example Three - Counting Regions

As a final example of the uses of morphology in binary images, it is possible to count the number of regions in an image using morphological operators. This method was first discussed by Levialdi, and uses six different structuring elements. The first four are used to erode the image, and were carefully chosen so as not to change the connectivity of the regions being eroded. The last two are used to count isolated '1' pixels; in **MAX** this is done using the '#' operator, and so these structuring elements will not be needed.

Figure 3 shows the four structuring elements, named L1 through L4. The initial count of regions is the number of isolated pixels in the input image A, and the image of iteration 0 is A:

$$count_0 = \#A$$

$$A_0 = A$$

The image of the next iteration is the union of the four erosions of the current image:

$$A_{n+1} = (A_n \ominus L_1) \cup (A_n \ominus L_2) \cup (A_n \ominus L_3) \cup (A_n \ominus L_4)$$

And the count for that iteration is the number of isolated pixels in that image:

$$count_{n+1} = \#A_{n+1}$$

The iteration stops when $\mathbf{A_n}$ becomes empty (all 0 pixels). The overall number of regions is the sum of all of the values $\mathbf{count_i}$. The **MAX** program that does this is:

```
// Count 8-connected regions.
image L1, L2, L3, L4, a, b;
int count;
begin
 L1 := {[2,2], [0,1], "0101"};
 L2 := {[2,2], [0,1], "0110"};
 L3 := {[2,2], [0,1], "1001"};
 L4 := {[2,2], [0,1], "1100"};
 do a << "$1";
 count := 0;
 loop
  count := #a + count;
  b := (a--L1) + (a--L2) + (a--L3) + (a--L4);
  exit when b = 0;
  a := b;
 end;
 message "Number of 8 regions is ";
 message count; message;
end;
```

This program counts eight regions in Figure 4, which is correct for 8-connected regions. It also counts two regions in Figure 3a, which is also correct. the algorithm for 4-connected regions is the same, but uses different structuring elements.

5. The Roughness Spectrum

First used as a measure of pore roughness in reservoir rock, the *roughness spectrum* [2] is essentially a count of lost pixels during opening of successive sizes. Given an initial image, count the connected regions. Then perform a binary opening, and count again; the number of objects lost is counted. Repeat, using consecutively larger openings, until no objects remain, and the result is the roughness spectrum.

The **MAX** program for the computation of the roughness spectrum is 63 lines, including annotation. The corresponding C code is 205 lines long, not counting the libraries. For a simple image of size 191x240 needing 5 iterations to complete (but actually computing the requested 20 iterations), the program requires

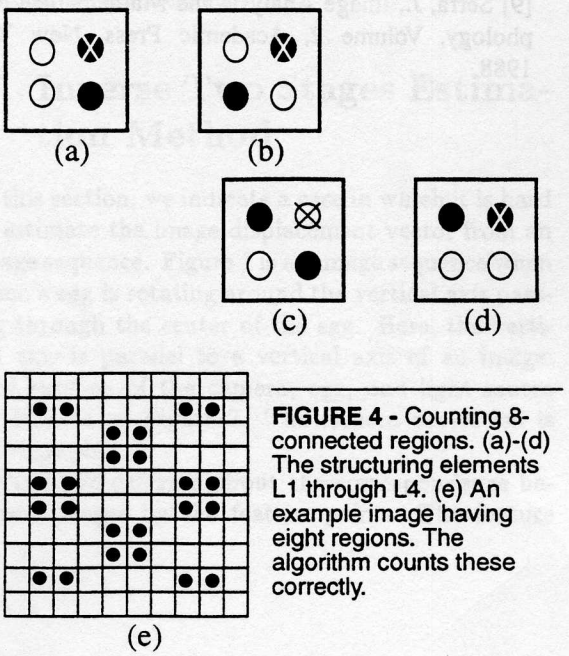

FIGURE 4 - Counting 8-connected regions. (a)-(d) The structuring elements L1 through L4. (e) An example image having eight regions. The algorithm counts these correctly.

358 CPU seconds to execute, which is far too long. However, the program took under an hour to write, and as a prototype it is, I feel, a success.

6. References

[1] Dougherty, E.R., An Introduction to Morphological Image Processing, SPIE Press, Bellingham, WA., 1992.

[2] Ehrlich, R., Crabtree, S.J., Kennedy, S.K., and Cannon, R.L., Petrographic Image Analysis, I. Analysis of Reservoir Pore Complexes, Journal of Sedimentary Petrology, Vol. 54. No. 4, Dec. 1984. Pp. 1365-1378.

[3] Giardina, C.R. and Dougherty, E.R., Morphological Methods in Image and Signal Processing, Prentice-Hall Inc, Englewood Cliffs, NJ. 1988.

[4] Gonzalez, R.C. and Woods, R.E., Digital Image Processing, Addison-Wesley Publishing Company, Reading, MA, 1992.

[5] Haralick, R.M., Sternberg, S.R., Zhuang, X., Image Analysis Using Mathematical Morphology, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. PAMI-9, No. 44, 1987. Pp. 532-550.

[6]Levialdi, S., On Shrinking Binary Picture Patterns, Communications of the ACM, Vol. 15, No. 1, Jan. 1972. Pp. 7-10.

[7] Parker, J.R. and Horsley, D., Grey Level Encoding of Openings and Closings, SPIE Vision Geometry II, Boston, MA. Sept. 9-10, 1993.

[8] Serra, J., Image Analysis and Mathematical Morphology, Academic Press, New York, 1982.

[9] Serra, J., Image Analysis and Mathematical Morphology, Volume 2, Academic Press, New York, 1988.