MICROMORPH®
Frequently Asked QUESTIONS

Serge Beucher, Dimitri GOROKHOVIK, Jean SERRA

Centre de Morphologie Mathématique

ECOLE NATIONALE SUPÉRIEURE DES MINES DE PARIS

MICROMORPH® in questions

Description of MICROMORPH®

The start-up environment of MICROMORPH® is defined by two files: ***mmorph.ini*** and ***init.mic***. We strongly advise you not to modify these two files before understanding thoroughly their content. The on-line help provides a detailed description of these files. In case ***mmorph.ini*** has been modified you can return to the initial version by copying this file from the CD-Rom to MICROMORPH® directory.

The virtual image processor that is the core of MICROMORPH® is configured to contain 10 8-bit greylevel memory-images (referenced as **g1** to **g10**), 10 16-bit binary memory-images (**b1** to **b10**) and 3 grey-level memory-images (**h1** to **h3**) at the start. The procedures and functions of MICROMORPH® are stored in a dictionary whose size is determined by ***mmorph.ini*** (it should be large enough for most of your applications. If this is not the case, it can be modified like most other features of the virtual processor).

Getting started

When MICROMORPH® is started several windows are opened : one that corresponds to the command interpreter, and four others for image display. How to use them?

The command interpreter

The command interpreter window allows you to enter MICROMORPH® commands from the keyboard and to start procedures. This window is made up of two parts : the lower part corresponds to the current command, while the upper part contains the log ("history") of the current session.

You can easily take an instruction from the upper to the lower (active) part by clicking the mouse. (**Important!** To enter any command, the interpreter window has to be active. The color of the bar at the top of the window generally indicates its status. To activate a window, you just have to click it).

Besides, this window can be edited with the ***edit*** function of the menu bar. The ***file*** function allows you to compile the routines written by using the programming language contained in a text file (see below).

Visualization

It is possible to display several images at the same time. When MICROMORPH® is started, 4 windows appear (2 binary and 2 greylevel), displaying the content of the 4 **g1**, **g2**, **b1** and **b2** memories, but other memories can be displayed. The management of the display windows can be done by using the **windows** submenu found in the main menu. The visualization can also be obtained by means of MICROMORPH® commands, such as **imdisplay**, **ds**, **dscol**, etc.

In order to load an image file into an image memory, you can use the MICROMORPH® command **imload**. You can also use the mouse; click the corresponding display window, and then click the ***file*** function of the main menu, and next the ***load*** function. The image directories appear where you can choose the image to be loaded into the selected memory. The reverse operation (to store a processing result into an image file for example) is executed in a similar way, the only difference is that ***save*** replaces ***load***.

The image displayed in the active window can also be put into the clipboard, by using the ***edit*** function of the main menu. The reverse operation is possible too. This allows to exchange images with other applications very rapidly (with word processing, or image enhancement programs, for example).

What kind of data does MICROMORPH® process?

The interpreter of MICROMORPH® language operates from the following types of data:

- integers,
- strings of characters,
- table of integers,
- images.

In this language:

- Floating point numbers are not supported (in mathematical morphology, this kind of computation is generally useless and slows operations).
- Integers are 32-bit-wide. They are well supported by arithmetic operations. One can allocate an integer from the global scope, from the local scope or from the dictionary.
- Character strings are supported to a lesser degree, some constraints are imposed when using them. They find their best use as the call-time parameters of functions and procedures.
- Arrays of integers can have up to three dimensions. They are allocated only from the dictionary, their size being limited only by the availability of memory space.
- Images are described by their size and depth. They are always defined and displayed in square raster (it is the *processing* that can be performed in square or hexagonal mode: try to apply, for example, the same **dil** operation to the same image, by modifying only the grid).

There exist also a number of derived types, that are also represented by means of integers:

- structuring elements,
- pixel values,
- pixel coordinates,
- directions.

How to code structuring elements?

The following code is designed to assign an integer to each subset of the unit-hexagon or -square. This code allows to introduce these subsets as structuring elements in ertain procedures (*e.g.* **hit-or-miss**). Thus, the neighborhood elements of a pixel are assigned the following weights:

in hexagonal raster	in square raster
64 2	256 2 4
32 1 4	128 1 8
16 8	64 32 16

A structuring element is coded by summing the weights of all its pixels. For example, the structuring element:

```
  +   +
*   +   *
  +   +
```

where " + " denotes the pixels is coded 91 = 64 + 2 + 1 + 16 + 8.

How to code directions?

The directions of the working raster are coded as follows:

in hexagonal raster			in square raster		
	6	1	8	1	2
5	0	2	7	0	3
	4	3	6	5	4

A direction represented by any value other than [0..6] in the hexagonal raster, and other than [0..8] in the square raster, is illegal, and sends an error message.

What about image depths?

The images are two-dimensional and can have three possible depths:
1-bit-per-pixel. They are called " binary images " and their values are 0 or 1;
8-bit-per-pixel. Each value is an unsigned 8-bit integer, in the range [0, +255];
16-bit-per-pixel. Each value is a signed 16-bit integer, in the range [-32768, +32767].

In the following **min** and **max,** denote the minimal and maximal values of the valid range, as indicated in the table below:

	min	**max**
binary images	0	1
8-bit images	0	255
16-bit images	-32 768	32 767

For any given image these values can be queried using the primitive functions **impixmin** and **impixmax**. Note that for 16-bit images, the values 0 and -32768, when negated, yield themselves. -32768 is the " infinity " value for this type of images, and is both positive and negative.

The functions that accept pixel values as input parameters — **imwritepix, imdrawline, imset, imcadd, imcsub, imcmul, imcdiv, imcopyngb, immask**, etc. — use the so-called generic pixel representation. The generic pixel representation is the signed 32-bit integer taking values from **min** through **max**.

In some operations, invalid values for such parameters are silently converted and no error is signaled. Therefore, it is left under the responsibility of the caller to supply valid data in the generic pixel representation. For example, when performing operations on binary images, any non-zero value in the generic pixel representation is considered as a representation of " 1 " and converted accordingly. Thus, performing **imset -1 im1** for a binary image **im1** produces the same result as **imset 1 im1**.

When operating on 8-bit images, negative values are converted to unsigned and truncated ones as necessary. Then performing **imset -1 im1** on an 8-bit image **im1** produces the same result as **imset 255 im1**.

When a primitive word is supposed to return or to use a pixel value for internal purposes, the latter is also represented by a signed 32-bit integer whose value is in the range from **min** to **max**. To give an example of an internal conversion of pixel values : adding a binary image to a greylevel one is performed as if two greylevel images were added, one of them having only 0 and +1 pixel values.

Is it possible to use colour images?

Colour images are not directly processed by MICROMORPH® as morphological operators are not usually defined for this kind of images (like most operators in image processing, mathematical morphology works with scalar and not with vector data such as those corresponding to colours). This is the reason why an error message will be prompted if you try to load a 24-bit or 32-bit BMP image (remind that 1, 8 or 16 bits are the only image depths allowed by MICROMORPH®). In order to process a colour image with MICROMORPH®, the useful information has to be extracted first. It may be either the luminance or any other scalar extracted from the representation space (hue, saturation, chrominance, etc.). However, these extraction facilities are not available within MICROMORPH®. There exist so many colour representations that the integration of conversion/extraction routines for colour components in MICROMORPH® would have involved a huge and useless work, since many software programmes already perform this task. Therefore you have to prepare the images to be processed with MICROMORPH® so that they represent scalar. This preparation is possible without quitting MICROMORPH® by calling external programmes (see how in this FAQ) and by transferring the images via the clipboard (this operation is also explained in this FAQ).

Conversely, one could think that the processing of 8-bit images (256 colours) by MICROMORPH® goes without difficulty. Indeed, the loading of such images into MICROMORPH® will generate no error. However, it is most probable that the results of the transformations will have no meaning. In this case, the numerical value associated with each pixel of the image corresponds to an index in a look-up table called ***colour palette***. This colour palette defines the set of the image colours. It is contained in a BMP image-file. However MICROMORPH® does not take this palette into account. Only the scalar value of the pixel is considered. The same value can correspond to very different colours according to the used palette. This is why the only palette that is meaningful with MICROMORPH® processing is the grey palette. In this case, the colour of a pixel of value n will be the RGB vector equal to (n,n,n), that is actually the grey value n. If you display an 8-bit colour image in MICROMORPH®, this image will be displayed as a greylevel image. Its aspect will give you a quite precise idea of the actual values used by MICROMORPH®. Make the test and you will, most probably, be surprised by the result!

How to use the colour palettes?

The image display in MICROMORPH® can be enhanced by the association of colour palettes. At the start-up, MICROMORPH® looks for the file containing the default colour palette. This file, called DEFAULT.PAL, must reside in the directory defined by the environment variable PAL in the MMORPH.INI configuration file. If the file is not found or has a wrong format, MICROMORPH® computes the internal palette with 256 grey levels, and uses it as the default one.

The default palette defines the initial look of all image displays. It is attached to all image windows and is referenced by “handle” 1.

One must distinguish images (which “know” nothing about palettes, see colour image processing in this FAQ), from their display windows. Only the latter can be attached palette 1 when calling **imdisplay** and receive another palette afterwards, by means of **imattach**. A palette becomes active only when the image is displayed. When pasting an image from the clipboard or loading it from disk, the palette stored in the source bitmap is never consulted. In no way can it affect the subsequent display of this image or other (again, see colour image processing).

Similarly, when saved to a disk or copied to the clipboard, the resulting bitmap image always gets the “greylevel” palette and not the one that is currently attached to the image display, if any. The attached palette is only a display feature within MICROMORPH® and is never exported.

After the image is displayed in a window, you can associate another palette with this window. In order to do this, you must have the necessary palette loaded by means of **imloadpal**. This function returns the “handle” to be used when calling **imattach**.

Once loaded, a palette is stored in MICROMORPH® memory until it is explicitly discarded, and can be attached to one or several windows. A palette can be discarded from memory at any moment by using **imdiscard**, whatever the number of its attachments. However, the default palette (number 1), cannot be discarded.

To detach the palette from a window without discarding it, just replace it by the default one

imattach 1 imin

It is exactly what happens when one discards the palette that is attached to one or several windows.

In 8-bit video mode (allowing for 256 colours maximum simultaneously), at any time there's only one window that correctly displays the colours, the one that is currently “active”. Consequently, to obtain the correct display, you have to “activate” the necessary window, by clicking it or its border.

In 16- or 24-bit and higher video modes there are no such limitations.

When a colour image is defined by three greytone images (RGB, HLS, and so on), this image can be processed by means of the Micromorph greytone transformations. However, it is not possible to directly display this colour image, unless a reduction of the colour palette to 256 colours is performed before. There exist, to achieve this, various pieces of software, performing at the same same a colour reduction and a definition of the corresponding colour palette. In most cases, the palette file contains a header followed by 256 colour vectors, each one made of three colour components. The header needs to be removed and the corrected file must be stored in a Micromorph "pal" file. Then, the image display is obtained by:

col in "palette" { assign the selected palette to image memory "in"}
imload in "reduced_image" { load in "in" the 256 colours image}

Moreover, various standard palettes are delivered with Micromorph. These files have the extension "pal" and are stored in the Micromorph main directory. They allow some colour displays, as "dscol" (color superposition of two images), "pscol" (labelling and colorisation of the various connected components of a binary image), "profile" (greytone image grey profile drawing), etc..

What are the working modes used by MICROMORPH®?

A mode is a processing choice, that is preserved throughout the instructions until explicitely changed. MICROMORPH® uses three major modes, which are independent of one another.

They are :

1 - choice of the raster, square or hexagonal;
2 - processing of the edge effects, standard or euclidean modes;
3 - size of the processed images.

The first two modes bear on a binary choice, where the default option, when starting a new session, is the hexagonal raster with standard treatment of edge effects.

The size of the processed images is expressed on the one hand by the image format, and, on the other hand, by the choice of a working window. By default, these two parameters are equal and their value is 256*256 pixels.

The choice of a mode is exclusive: processing images in square and hexagonal rasters simultaneously is impossible, for example. In this respect, the depth of images does not constitute a mode, one can work with binary and greylevel images simultaneously.

How to indicate and/or modify the size of images?

The horizontal and vertical dimensions of images are two modes controlled by the primitive word **iminit**, or by the composite word **format** as well. The latter command also allocates a certain number of greytone of binary default images. The minimum size along the X axis is 128 pixels. It will always be a multiple of 32. The minimum size along the Y axis is 2 and will always be even. Theoretically, the image dimensions can extend to 16384 x 16384. In practice, the virtual memory available may reduce this maximum size. When an image is called that is bigger than the current format, it is the upper left part of the image that is loaded.

The working zone inside the image is also a mode, controlled by the primitive word **imsetwindow**. By default, the dimensions of the working window are those of the image.

How to read pixel coordinates?

The coordinates of the pixel of the upper left corner are (0,0). The X coordinate is incremented from left to right, and Y from top to bottom. Images are assumed to be entirely located in the first quadrant of the coordinate system. Consequently, any negative coordinate referring to a non existing pixel will be rejected or converted, depending on the current operation (see **imwritepix, imreadpix**).

Square or hexagonal raster?

The images are processed in square or hexagonal raster. The choice of the raster is ruled by the primitive word **imsetgrid**:

imsetgrid 0 sets the square raster for all images;
imsetgrid 1 sets the hexagonal raster for all images

The type of grid only affects the execution of six neighbourhood operations, namely : **imcopyngb, iminfngb, imsupngb, imdiffngb, imbuildngb** and **imhitormiss**. The type of grid is more precisely associated with these operations than to images themselves. Thus, all the images allocated after a call to **imsetgrid** will have the raster type specified by this call, and the next call to **imsetgrid** will change the raster type for the images already allocated.

For both rasters, the Y coordinate increases from top to bottom; the X coordinate increases from left to righ ; the upper left pixel of an image has the coordinates (0,0). In the case of the hexagonal raster, each image line has the associated parity: the lines with the even Y coordinate, starting from “0”, have even parity, the others have odd parity.This has nothing to do with line length, but corresponds to the following layout:

line 0 : * * * * * * *
line 1 : * * * * * *
line 2 : * * * * * * *
etc...

The type of raster can be queried at any time using the word **raster**.

How to handle edge effects?

Every image, whatever its type, has an outer border : one line at the top and the bottom of the image, and one 1-pixel-wide column at the left and one at the right. This border is not visible on displays. It only affects the functioning of the five hood operations: **iminfngb, imsupngb, imdiffngb, imbuildngb, imhitormiss.** When executed, each operation presets the outer border as appropriate (however, there is no specific values stored for the edge or outer border).

The preset value of the outer border is ruled by the primitive word **imsetedge**, according to the following table:

operation	value of **edge**	
	0	1
iminfngb	**min**	**max**
imsupngb	**min**	**min**
imdiffngb	**min**	**min**
imbuildngb	**min**	**min**
imhitormiss	**min**	not applicable

where **min** and **max** denote respectively the minimum and maximum pixel values for the current image type, that is, 0 and 1 for binary images, 0 and 255 for 8-bit images, and -32768 and + 32767 for 16-bit images.

The effect of **imsetedge** applies to the whole image or to the restricted zone of interest defined by means of **imsetwindow**. In the latter case, if the processing zone is strictly smaller than the image size, the pixels surrounding it are never modified.

The current value of the edge effect can be queried at any time using the function **edge.** Note that **imsetedge** and **imsetgrid** are two independent operations.

How to execute a procedure or a function?

The words of MICROMORPH® are divided into two categories. Those returning a number are called "functions", the others are called "procedures". Both categories can be executed in two ways :

1 - They can be entered from the keyboard into the command window, then press the **enter** key ;

2 - It is possible to create new procedures or functions, which implies to name them. A procedure is written under the form of a text file using any text editor, and then compiled. Its name is then integrated as an additional word into the Dictionary. If the same word is already used, the new definition automatically replaces the previous one.

How to create a new procedure or function?

A very interesting feature of MICROMORPH® is that it allows you to define your own routines and add them to the dictionary, by using already available procedures and functions. This is a chained programming structure. The simplest way to define new words is to write their definition in a text file with your favourite text editor (Windows® notepad, WordPad word processor or any other word processor that enables you to store a simple text file will do).

Use **deproc** to create a new procedure, and **defunc** to create a new function. As the same comments apply to both procedures, we will only deal with the first one in details.

Syntax

deproc procname **pre-params** procname **post-params**
syntax **"**useful comments**"**
  sequence of instructions
end

Effect

Defines a new procedure whose name is *procname*, equipped with pre-parameters or post-parameters or both.

The name of the word to be created must be indicated twice: the first time, immediately after **deproc,** in order to declare it, and the second time, as a separator, between the list of pre-parameters and that of post-parameters. This presentation is described by the **syntax** operator; this is the reason why this operator is indispensable just after the definition of the word.

The word **syntax** also allows to introduce comments, which are then inserted into the Dictionary. If you don't want to, you can use the short version by entering **syntax '' ''**.

How to introduce parameters into a procedure?

The parameters of a function (or of a procedure) can be passed “by value” or “by reference”. This is illustrated below with a simple example. Let us take a procedure P to compute the double of a numerical input value and print the result on screen:

```
deproc P P v
syntax "P value -> prints the double of value"
v := ( 2 * v )
print v
```

end

We apply this procedure to a variable A that we have declared by **int A ;** and to which we assign value 5: **A := 5.** In this case, **P A** prints "10" on screen. **P 5** gives the same result. Now, if we print A by **print A**, we notice that the value remains unchanged and still equals 5. We say that variable *A* is passed by ***value***, since its value is only copied when calling the procedure.

Let us define again the same procedure, but denoted by parameter *v to specify that it is passed by ***reference:***

```
deproc P P *v
syntax "P  value -> prints the double of value"
*v := ( 2 * *v )
print *v
end
```

As before, **P A** prints ‘’10’’ on screen. But this time, if we enter **print A,** we see that the value of *A* is now 10. Besides, executing **P 5** now produces an error message “ variable expected in P ”. This means that the value of *A* is not copied to *v* anymore. Its address has passed to *P*, where it is used as the address of *v*.

Example

```
deproc env env
syntax "env (no parameters) -> prints some vital values"
print [ "edge = " edge ]
print [ "raster = " raster ]
print [ "connectivity = " connectivity ]
end
```

How to compile a file?

Once new procedures or functions are created (you can write several routines in the same file), you just have to compile the file using the ***file/Compile*** instruction of the menu bar in the command interpreter window. **Don’t forget to save the file before compiling it**. If there are no errors in your source code, you will see the new routines added to the dictionary (this can be checked in the Dictionary window). From this moment, these new routines (words) can be executed from the interpreter or used to compose more complex words.

The file can also be compiled by using the **comp** procedure :

Syntax

comp "filename**"**

Effect

Compiles the file *filename*. Compiling a file consists in interpreting its contents as if it were typed from the interpreter’s prompt. Script files must have a size lesser than 65536 bytes. Files containing lines of a length greater than 255 characters cannot be compiled properly, the offending lines are silently truncated.

How to add new routines to MICROMORPH start-up?

In order to avoid the manual compiling of your own source files each time you start MICROMORPH®, you can insert a command in the ***com.mic*** file to compile them automatically. This file contains all the files that are compiled automatically by MICROMORPH® at start-up. In fact, MICROMORPH® executes the ***init.mic*** file, which in turn compiles the ***com.mic*** file, which compiles the other files. Suppose that the source file you have created is called **myprog.mic**. Then, simply add, at the bottom of ***com.mic***, the line:

comp "myprog"

and the routines defined in **myprog.mic** will be automatically compiled each time you start MICROMORPH®.

You can also modify the initial contents of MICROMORPH® dictionary by editing the file ***com.mic***. However, be sure you know exactly the contents of the source files before modifying the dictionary in order to respect the proper chaining of operators. Suppressing a file may produce compiling errors if the following files cannot find the procedures that are indispensable to their execution.

How to insert an on-line help into a procedure or function?

For advanced users !

It is possible to execute an external programme by means of the word **syntax.** To do so, first define the external programme. This is done by a string between quotation marks beginning with an exclamation mark, continuing with the name of the programme, followed by any necessary parameters. Then, compile the procedure. To start the programme you have defined, just click the corresponding procedure in the dictionary window. For example, when writing a programme whose source code is in **file.mic**, the latter can be accessed at any time by entering:

syntax ‘’ ! notepad fichier.mic ‘’

Here is another example: to call Windows on-line help using the word **syntax**, a small executable programme is provided with MICROMORPH®, **whelpdrv.exe:**

syntax ‘’ ! whelpdrv mmorph.hlp imabsmax ‘’

You can replace the help file by any help file you may have created and comment the procedure or function you have just defined. You can also use other means, if writing a help file seems too difficult. You can describe and comment your procedure in a HTML file (named **myprog.htm** for example) and display it when you click the name of the procedure in the dictionary by using the instruction:

syntax ‘’ ! iexplore myprog.htm’’

if you use Internet Explorer as your web navigator and if the path has been specified at the start-up.

How to call an external programme from MICROMORPH®?

You can start a Windows application from MICROMORPH® using the command **!** followed by the name of the programme. This command starts the execution of the application and immediately returns to the interpreter. It never returns any value. The interpreter has no means to detect the termination of an application called via " **!** ". This means that MICROMORPH® instructions following **!** will be immediately executed. If you expect some return (image, value...) from the external programme, you have to insert a break in MICROMORPH® programme to make sure that it resumes only after the external application is over. This command can only start applications existing as executable files on disk (it cannot for example execute commands contained in command.com. If the file does not happen to be in one of the PATH directories, you have to supply its full pathname, for example:

! c:\user\mapsym.bat

How to use the clipboard?

Windows clipboard can be used with MICROMORPH® in order to import images from external programmes or to export MICROMORPH® images. This facility is particularly interesting to retrieve images from capture or graphical retouching facilities, or else to export images to programmes with processing functions not present in MICROMORPH® (for example separation or association of colour image components). The clipboard can be used via the ***edit*** menu, or via **imclipcopy** and **imclippaste** which can be entered in the command interpreter or in the definition of a procedure or function.
Important ! You can only transfer Windows DIB formatted images (8-bit BMP format) otherwise you will get an error message. Besides, the colour palette of the image is lost whatever the way (in or out of MICROMORPH®). The only palette that is preserved is the **greylevel** one. (See sections on colour images and palettes).

How to use logical loops in a procedure?

MICROMORPH® includes the logical loops **for**, **if** and **while**. Their syntax is the following :

for

Syntax
for expression1 **to** expression2 **do** sequence of instructions **end**

Effect
Executes *sequence-of-instructions* in the loop (*expression2 - expression1 + 1*) times. Both expressions are evaluated only once, before entering the loop. Any loop may be interrupted by Control-C. The words **do** and **end** are indispensable, even when the *sequence-of-instructions* consists of only one word. Besides, the word **do** must be on the same line as **for**.

Example

```
for 1 to 0 do print "nothing" end
int a N ;
a := 1 N := 20
for a to N do
   print "N times"
end
```

if

Syntax
if expression **then** sequence of operators **end**
if expression **then** sequence of operators_1 **else** sequence of operators_2 **end**
if expression **then else** sequence of operators **end**

Effect
Conditional operator

Example
if (a = 1) **then** print "a equals 1"
else print "a does not equal 1"
end

while

Syntax
while *expression* **do** *sequence of operators* **end**

Effect
Executes *sequence of operators* as long as the value of *expression* is not 0. Any loop, even an infinite one, can be interrupted by Control-C. **do** and **end** are indispensable and must always be present in the construction, even if the *sequence of operators* consists of only one operator. Besides, the word **do** must be on the same line as **while**.

Examples

1/ **while** 1 **do** print "infinite" **end** { prints “ infinite ” indefinitely }

2/
```
int a N ;
a := 0
N := 20
while (a < N ) do
print cat ["a = " a ]
a := ( a + 1 )
end
```
{the procedure prints 20 times ‘’a = ‘’ followed by the value of a}

3/
```
int w2 ; w2 := imalloc 8 imcopy g1 w2
while ( imvolume w2 ) do
{ sequence of operations }
end
```
{the sequence of operations is iterated until the volume (the area) of w2 equals 0}

How to allocate 2D image spaces in a procedure?

It is often necessary to allocate memory spaces to store processing images. To do so, use **imalloc** procedure.

Syntax
imalloc pixdepth

Effect
Allocates an image of a specified pixel depth and returns the handle of the new image. Valid pixel depths are 1, 8 and 16. The image is created with the dimensions specified in the last call to **iminit**. The depth of an allocated image can be obtained at any moment by calling **imdepth**. All the images allocated within a procedure defined using **deproc** or **defunc** will be automatically unallocated when returning from this procedure. The only exception occurs where there is a call to **iminit** in the procedure. Then, all the images allocated below this call are kept (this feature is used in the **format** procedure).

Example

```
deproc top top s d sz                    { s and d are not introduced as memory spaces }
syntax "grad source destination size : top-hat by opening"
    int u v ;
    u := imalloc 8     v := imalloc 1
    open s u  sz
    imdiff s u d
   imthresh d 0 5 v
   imdisplay v " thresholded tophat "
 end                                     { the spaces u and v are disallocated, image v disappears from screen}
```

How to allocate 3D image spaces in a procedure?

The 3D procedures defined in files "3dutil.mic" and "3dproces.mic" require the allocation of memory spaces for the images belonging to the sequences under study. But the "init.mic" file already presets a first bunch of allocations: 10 8-bit image memories, 10 binary image memories and 3 16-bit image memories. Therefore, if we enter:

```
int i ;
i := imalloc 1
```

a 1-bit image memory space is allocated, its label been equal to 24. To verify this and display this memory, simply type:

```
ds 24
```

More generally, if we need 60 8-bit memories for instance, for a 30 images initial sequence and for 30 transformed images, then, before using seqload, we shall type:

```
int i ;
for 1 to 60 do
i := imalloc 8
end
```

As a result, 60 image memories, labelled from 24 to 83, will be allocated.
If now we enter:

```
Seqload "c:\studies\seq\" 8 30 10
```

the following operations will be performed:

```
imload 30 "c:\studies\seq\8.bmp"
imload 31 "c:\studies\seq\9.bmp"
....
imload 39 "c:\studies\seq\17.bmp"
```

If you write your own 3D transformations, it is suggested that you create memory spaces inside your written procedures, so that these spaces can be automatically removed at the end of the procedure, avoiding then to occupy a great amount of working memory space inside the computer.

How to display a result?

The basic display statement is **print**. when associated with **output**, it defines where the result is displayed.

Syntax
print expression
print **[** expression1 ... expressionN **]**

Effect
This procedure transfers to the output destination one (first form) or several values of concatenated expressions (second form). The destination can be the screen or the file on disk (see the word **output**).

Example

```
print "character string" { displays ‘’character string ’’ }
int i; print i
print [ "i = " i ] { displays : ‘’i := ‘’ followed by the value of i }
```

How to store results, and print graphics?

As a general rule for MICROMORPH® functions, values are only stored in a variable, and displayed by means of **print**. If you want to record them, the function must be preceded by the opening and the closing of a file. Thus, if you want to know the evolution of the perimeter when eroding image b1, write:

```
output "c:\b1.dat"
while  (area b1) do
print perim b1
ero b1 b1 1
end
output " "
```

However, most functions of the section 15 of the reference manual (curves) offer a shortcut, as they admit a file name as input variable. For example, for the granulometry of lung1 and lung2 images, 1 by 1 until step 25, write, after loading images in g1 and in g2:

```
granul  g1  1  25 "lung1.dat"          granul  g2  1  25  "lung2.dat"
```

The values are then recorded in the files *lung1.dat* and *lung2.dat*. They can be used for any later processing. In particular, if you want to print them graphically, you can use an external programme such as ***gnuplot*** where the following sequence of instructions:

set xlabel "SIZE" 0 set ylabel "AMOUNT" 0
set title "Grey size distribution (by openings)"
plot "c:\wmmorph\lung1.dat" with linespoint , "c:\wmmorph\lung2.dat" with linespoint

produces two granulometric curves on the same graphics.

What is the Dictionary?

The dictionary contains the list of all available words (functions and procedures). Procedures and functions are compiled at the start-up. They are present in the software core or contained in the files with **.mic** extension. The initialization of MICROMORPH® provides a considerable number of procedures. In fact, there exist two types of words. The primitive words constitute the basic core of MICROMORPH®. They cannot be modified. In addition to this basic core, comes the collection of the composite words already prepared by the user. The composite words are elaborated from the primitive words, or from simpler composite words, as indicated in the section *«How to create a procedure?»*. You can also enrich the dictionary, and customize it with your own words in a very simple way. You just have to compile the corresponding source files. The programming structure of MICROMORPH® language is rather simple and the addition of new transformations can be done without difficulty.

To open the dictionary, click the corresponding icon, which is always visible. Then click a word, and its syntax will appear, or in case of a primitive word or one that is associated a help file (see the corresponding section), a description of its effect, its working features (modes, image depths,...), sometimes an example and a list of close words, are provided. The working features are specified by the following conventions :

raster : 1	hexagonal raster only;
raster : 0	square raster only;
raster : 1,0	square or hexagonal raster ;
raster : [1],0	automatic switch to square raster for the current procedure, and return to the previous raster (changing whatever is necessary for *raster* : 0, [1]).
edge : 1	standard mode only;
: 0	Euclidean mode only, intrinsic (increasing op.) or transitive (thinning);
edge : [1],0	automatic switch to **imsetedge** = 0 for the current procedure, and return to the previous state (changing whatever is necessary for *edge* : [0], 1).
depth :	bin ; grey ; bin to grey, etc... this term is not associated with a specific mode, as *raster* or *edge*, but indicates the depth in bits of the processed images.

For example, the *same* **dil** procedure performs, according to the key values, a hexagonal or square dilation, in standard or euclidean mode. And it is automatically set to binary or greylevel mode depending on whether the processed image is a binary or greylevel image. Finally, as the dictionary classifies the words in alphabetical order because this order is

universal (but not the most appropriate for finding them), we give below a thematic index for both series of primitive and composite words.

THEMATIC LIST OF MICROMORPH® WORDS

I - List of the primitive words

1 - GENERAL-PURPOSE IMAGE HANDLING

imalloc, imfree	allocate, deallocate a 1-, 8- or 16-bit image
imclippaste, imclipcopy	get/put an image on the Windows clipboard as bitmap
imcopy, imcopyhibyte, imcopylobyte, imcopyplane	copy or insert an image into another
imdepth, impixmin, impixmax, imparam	query image and processing zone dimensions, pixel depth
imdrawline	draw a segment in an image
iminit, imsetwindow	set image and processing zone dimensions
imload, imsave	load, store an image in a.BMP file
imwritepix, imreadpix	write, read a pixel in an image

2 - ARITHMETIC AND OTHER POINTWISE OPERATIONS

imabs	convert pixels of 16-bit image to absolute values
imabsmax, imabsmin	find a maximum/minimum among all the pixels in the image
imadd	add two images together
imcadd	add a constant to an image
imcdiv	divide an image by a constant
imcmul	multiply an image by a constant
imcompare	compare two images
imcsub	subtract a constant from an image
iminv	invert an image
immask	construct a greylevel image from a binary one
imor, imand, imdiff, imxor	boolean operations on binary images
imset	fill an image with a value
imsub	subtract an image from another
imsup, iminf	find the maximum or the minimum of two images
imthresh	threshold an image
imvolume	calculate the sum of all the pixel values in an image

3 - NEIGHBOURHOOD OPERATIONS

imbuildngb	elementary reconstruction
imconvolve	image convolution with nucleus specified in a file
imcopyngb	translate an image in one of the raster directions
imdiffngb	set difference with a neighbour in a given direction
imhitormiss	«hit-or-miss» operation
imsupngb, iminfngb	elementary dilation and erosion

4 - OTHER

!	execute an executable file
?terminal, readkey	keyboard reading operations
comp	compile a script file
del	delete a definition from the MicroMorph's dictionary
edge, imsetedge	query or set mode of image border treatment
grid, imsetgrid	query or set image raster type
imdisplay, imunlock, imshut	
imlock, imscale	means of image display and update once displayed
imloadpal, imattach, imdiscard	color palette handling for colored image display
imtrace, imtracend, imgetcursor	means of reading mouse position in the image
input, output, read, print, eof	input/output along with the means of its redirection to file/screen
makeerror	produce an error message box
randomize, random	random number generation
serotate, setranspose	handle a structuring element
sound	make a beep

II - List of composed procedures or functions

1 - UTILITIES

clr	image clear
power	arithmetical power function
div	(arith) division rounded to the nearest integer
abs	(arith) absolute value
inside, inferior	A is included in B, f≤g
translate	translation
div2	division of image by 2
mean	mean of two images
incrust	inserts a part of one image into another
bintogrey	greytone version of a binary image
masksup (equal)	mask of the domain $g_1 > g_2$ ($g_1 \geq g_2$)
greymask	constant value in the given mask
s4rotate	rotation of structuring element by 90°
greyabs	absolute value of an image
format	Micromorph images format setup
adjust	interactive image thresholding
ngbnb	returns the number of neighbors of a point in the grid
dirtranspose	computes a transposed direction
clean	zeroing the low values
immean	mean grey vlue of an image
getcoords	gets the coordinates of a point
gandb	intersection between a binary and a greytone image
stop	stop and waits for a key press
stretch	stretchs the grey range of an image
delta	mask of the differences between two images
half	reduces by 2 the size of an image

2 - DISPLAY TOOLS

ds dscol	display shortcuts
shadow shadow2	shadowing operators
clrcol	deactivates a color palette
col	affects a color palette
pscol	color labelling
cp	copy an image with its palette
cppal	copy a palette
refresh	displays an image above the others
profile	grey profile display
coldisplay	color display

- EROSIONS AND DILATIONS

3-1 basic erosions and dilations

dil, ero	dilation, erosion, by a square or hexagon
dirdil, direro	dilation, erosion by a segment
minidil, miniero	dilation, erosion by a little square or triangle
distance	distance function
dbldil, dblero	dilation, erosion by a pair of points
cont, sq4cont	internal contour of an image (field)
gradient, sobel	morphological gradient and Sobel gradient

3-2 other dilations and erosions

isodil, isoero	isotropic dilation, erosion
isodist	isotropic distance function
conedil, cyldil	dilation by a cone, a cylinder
crossdil	dilation of a set by another set
rhombodil, rhomboero	dilation, erosion by a rhombododecagon
diamdil, diamero	dilation, erosion by a diamond
ringdil1, ringdil2	dilation by a ring or by the summits of an hexagon
(intermediary routines)	polygonal dilations

4 - RANK OPERATORS

hexrank	hexagonal rank operator
median	median filter
binsegmi	intersection of dil. by segments

5 - CONVOLUTIONS

vgauss1, hgauss1, gauss1	gaussian filters (vertical, horizontal and isotropic of size 1)
gauss	gaussian filter of size n

6 - OPENINGS, CLOSINGS

6-1 basic openings, closings

open, close	opening, closing by dil and ero
diropen, dirclose	opening, closing by dirdil and direro
lineopen, lineclose	union (intersection) of the above
openth, closeth	top-hat by opening, closing
lineopenth, linecloseth	top-hat by lineopen, by lineclose

6-2 other openings, closings

isopen, isoclose	isotropic opening, closing
infopen, supclose	limit of inf (sup) of directional openings (closings)
miniopen, miniclose	opening, closing by minidil, miniero
regrad, pregrad	regularized gradients
stopen	inf of directional openings

7 - GEODESY AND CONNECTIVITY

gdsdil, gdsero	geodesic dilation, erosion
build	binary reconstruction
buildopen, buildclose	opening, closing by reconstruction
areaopen	opening according to the area
ringbuild	reconstruction by a ring
gdsdist	geodesic distance
recons	binary reconstruction (similar to build)
levelling	levelling transform

8 - APPLICATIONS OF GEODESY

edgeoff	deletes the grains that touch the border
fgrain	extracts the first particle
border	internal contour of the field
clohole	filling the internal pores
minima, maxima	extrema of a function
extmaxima, extminima	extended extrema
swamping	modification of the homotopy by markers
extrema	extrema of an image inside a mask
grainclose	grain smoothing and closing
indivaf	individual closing of particles
dynamics	dynamics of the maxima or minima of an image

9 - FILTERS

9-1 main filters

af	alternating filter
asf, fullasf	alternating sequential filter (2 types of progression)
lineaf, lineasf	alternating sequential filter with lineopen
automed, centre	automedian operator, morphological centre
contrast	contrast for isoero-isodil, isopen-isoclose
contrasth	contrast for top-hat

9-2 other filters

minifilt	mini alternating filter
isaf, isasf	isotropic alternating filter, alternating sequential filter
buildaf, buildasf	alternating filter, alternating sequential, by reconstruction
closoropen	toggle between opening and closing
minibuildaf	filter by reconstruction using miniopen and miniclose
binmiddle	binary median set
grmiddle	**grey median image**

10 - MAXIMAL BALLS AND SKELETON

binopenskel	skeleton by maximal squares (hexagons)
binultim	ultimate erosion

centroid	centroid (last eroded set)
condbis	conditional bisector

11 - THINNINGS, THICKENINGS

11-1 binary

(gds)thickturn, (-)thinturn	basic (geodesic) thinning, thickening, cycle
(gds)thick, (gds)thin	thinning, thickening (until idempotence)
Dthick, Lthick, Mthick	homotopic thickening
Dthin, Lthin, Mthin	homotopic thinning
endpoints	ending points
multpoints	multiple points
gdscentre	geodesic centre

11-2 greytone

greyseero, greysedil	erosion, dilation by se
greythickstep, greythinstep	basic thickening, thinning
dirgradient	directional gradient
vectgrad0	complete coarse gradient
gradvect	true vector gradient
mainthin	thinning and clipping
greyhmt	greytone hit-or-miss
greythickturn	greytone basic thickening
greythick	greytone thickening

12 - WATERSHED TRANSFORMATION

clip	clipping
(gds)skiz	skeleton by influence zone
threshwshed	watershed by thresholding
mwshed, wshed	watershed, with or without markers

13 - SEGMENTATION

mgradwshed, gradwshed	watershed, with or without markers, of gradient
shapeseg	segmentation by distance function
smoothcnc, jumpcnc, jump	contrast extraction
mosaic	mosaic image
wfall	waterfall transformation
kheops	pyramid of mosaic images

14 - MEASURES

14-1 basic measures

diameter	diameter variation
digperim, perim	digital and Euclidean perimeters
cnumber	connexity number
binh(v)ferret	Ferret's diameter

14-2 other measures

var0, var1, var2	variances of order 0, 1 and 2
rug	roughness
mse	mean quadratic difference
bincount	number of connected components
flatcount	number of flat zones
flatzone	area of the flat zones

15 - CURVES

bincov	binary covariance
vario1, vario2	order 1 and 2 variograms
modcont	module of continuity
isogranul , granul	granulometry
covar	covariance
pvonl	measure of the binary linear erosion
p1vonl	binary linear granulometry in number
p2vonl	binary linear granulometry in measure

16 - RANDOM SIMULATIONS

16-1 boolean simulations

point, points, regpoints	random points
isobool, isobool2	simulation of Boolean isotropic sets
elbool, tribool, dropbool	simulation of Boolean anisotropy sets
rose, hard, disjoint, flake	simulation of hierarchies
rocky	rock bases

16.2 lines and partitions

lines, diags	Poisson lines
nestlines	Hierarchy of Poisson lines
steps	Poisson strips

17. GRAPHS

Gero, **Gdil**	erosion, dilation on graphs
Gopen, **Gclose**	opening, closing on graphs
setborder	outlines borders in a grey image
Gbuild	planar graph reconstruction
label	generates a label image

18. 3DUTILITIES

Seqload	loads a sequence
Seqsave	saves a sequence
Seqclr	clears a sequence
Seqset	sets a sequence to a given value
Seqcopy	copies a sequence
Seqinv	inverts a sequence

Seqadd	adds all the images of a sequence
Seqcadd	adds a constant value to a sequence
Seqseqadd	adds two sequences
Seqcsub	subtracts a constant value from a sequence
Seqseqsub	subtracts two sequences
Seqcmul	multiplies a constant to a sequence
Seqmean	mean image of a sequence
Seqseqmean	computes the mean of two sequences
Seqdiff	module of the difference between two sequences
Seqvolume	volume of a sequence
Seqinf	inf a all the images of a sequence
Seqseqinf	inf between two sequences
Seqsup	sup of all the images of a sequence
Seqseqsup	sup between two sequences
Seqbuild	reconstruction of a sequence from a marker sequence
Seqthresh	sequence grey thresholding
Seqgrad	gradient of a sequence
Seqhist	histogram of a sequence
Scroll, Scroll2, **Scroll3**	sequence scrollings
Visu	perspective display

19. 3D PROCEDURES

ero3D, dil3D	3D erosion and dilation
open3D, **close3D**	3D opening, closing
gradient3D	3D gradient
build3D	3D reconstruction
buildopen3D, **buildclose3D**	3D opening and closing by reconstruction
openth3D, **closeth3D**	3D top-hats
maxima3D, **minima3D**	3D extrema
af3D, **asf3D**	3D alternate sequential filters
buildaf3D, **buildasf3D**	3D AS filters by reconstruction
timdil, **timero**	linear erosion and dilation along the time axis
timopen, **timclose**	linear opening and closing along the time axis

20. MOUSE

fill	draws and fills a polygon
dline	draws lines
delobj	removes connected components
pointinfo	coordinates and grey value of a point
binpointer	points and extracts a particle
objinfo	area of a pointed particle
brush	brush function