DAA/LANGLEY
NAG 1-538

IN-61

ICL - The Image Composition Language

63709
37P

James D. Foley
Won Chul Kim

Institute for Information Science and Technology
Department of Electrical Engineering and Computer Science
School of Engineering and Applied Science
The George Washington University
Washington, D.C. 20052

September 1986

Report GWU-IIST-86-26

This work was supported by the Applied Mathematical Sciences Program, Office of Basic Energy Sciences, DOE, grant DE-AS05-83ER13122, and NASA-Langley Research Center, grant NAG-1-538 632200.

Abstract

The Image Composition Language provides a convenient way for programmers of interactive graphics application programs to define how the video look-up table of a raster display system is to be loaded. ICL allows one or several images stored in the frame buffer to be combined in a variety of ways. ICL treats these images as variables, and provides arithemtic, relational, and conditional operators to combine the images, scalar variables, and constants in image composition expressions such as

old_image * *t* + *new_image* * (1.0 – *t*), and
if *image*.r > 0.5 **then** *image* - {0.5, 0, 0} **else** *image* **endif.**

The objective of ICL is to provide programmers with a simple way to compose images, to relieve the tedium usually associated with loading the video look-up table to obtain desired results.

ICL - The Image Composition Language

1. Introduction

Programming of interactive graphics applications is a tedious and expensive process. Many details of interaction techniques, information presentation, display organization, etc. must be considered. Long sequences of subroutine calls to graphics subroutine packages are often needed to carry out relatively simple operations. As the cost of computer graphics equipment continues to drop, the cost of programming becomes more and more of a barrier to development of applications.

Why is graphics programming expensive? There are several reasons. First, the level of individual semantic units found in typical graphics subroutine packages such as the Core [GSPC79] and GKS [GKS84] are quite low, requiring multiple subroutine calls to carry out relatively simple operations. This is why higher-level implementation tools, such as User Interface Management Systems [OLSE84], dynamic hierarchical graphics packages [SHUE86], and graphics packages integrated with data base management systems [GARR82] are attractive. In some sense, working with traditional graphics package tools is akin to programming with rudimentary procedural languages, while working with higher-level tools is analogous to the so-called fourth-generation tools of program generators, query languages, etc.

The second problem is that calls to a subroutine package are often not the most useful way to express semantics, as illustrated by the two following ways to express arithmetic operations:

```
Add (a, b, c)           { a + b -> c        }
Add (e, f, g )          { e + f -> g        }
Mul (c, g, result )     { c * g -> result   }

result  :=  ( a + b ) * ( e + f  )
```

Clearly the programming language is preferable, in that the reader can much more quickly grasp the meaning of the assignment statement.

We believe that providing programming language tools for graphics application programmers can enhance their productivity. We have developed a language to facilitate one aspect of the work faced by application programmers - loading the video look-up table (LUT) of a raster display system in order to modify and/or combine multiple images stored in sets of planes in the display's frame buffer. With the Image Composition Language (ICL), the programmer writes composition expressions made up of arithmetic, relational, and logical operators which combine images, scalars, and literals. Each composition expression is associated with an image frame, to indicate a display region in which the expression is to be applied and hence in which the viewer will see the image created by the composition expression. Each image has its own logical look-up table, entries of which have either a color or a special value, **transparent**, to signify that pixels with the corresponding pixel value are to be treated as transparent when combined with pixels from other images.

The compiler for ICL, called the Look-Up Table Compiler (LUTC), automatically computes the appropriate look-up table contents needed to achieve the effect specified in the composition expression. The LUTC is implemented as a run-time subroutine library, and is written in Modula-2. Composition expressions are passed to the LUTC as character strings, after which they are compiled into look-up table contents.

ICL can be used to implement some common image processing, movie-making, and window manager operations. Examples from some of these are used throughout the paper to explain ICL concepts. ICL assumes that the refresh buffer planes have already been loaded with the images which are to be operated upon. The application program would normally do this using a graphics package such as Core or GKS, or a hidden surface removal and rendering package.

The ICL concept was first developed as part of a conceptual model for raster system architectures [ACQU82] which integrated a number of traditional graphics concepts with the newer set of raster concepts. A language for operating on images was developed as part of that work [ACQU84]. A refinement of that language is reported on in this paper.

The concepts of the ICL and LUTC can be extended to raster display architectures which provide a multi-bit RasterOp instruction cabable of performing arithmetic and boolean operations [PRES86].

2. Previous work

Two other related works are reported in the literature. The first is Guibas and Stolfi's elegant "Bitmap Calculus" and corresponding language MUMBLE [GUIB82] for manipulating bitmaps. The language provides boolean, relational, bitstring, conditional, assignment, and transformation operators on bitmaps and scalars. Also described are a compiler for processing the language, and a number of interesting and useful bitmap algorithms. The compiler generates sequences of raster-op and other related operations for execution in a single-address space raster display architecture.

Because the target machines and underlying objectives for the MUMBLE and ICL languages differ, the languages themselves differ in scope. MUMBLE provides expressions and flow of control; ICL does not. MUMBLE permits arbitrary-size bitmaps; ICL does not. The output of ICL is the (typically 1024) look-up table entries to evaluate an expression; MUMBLE output is a sequence of RasterOp and conventional instructions. With ICL, the precision of arithmetic results is limited only by the width of the LUT, while with MUMBLE, the precision is limited by the number of bits per pixel. With MUMBLE, the bit map variable to which an expression is assigned can be further manipulated, while the results of ICL expresions are only available visually on the view surface. In MUMBLE there is no explicit notion that a bit map may have one or more pixel values which mean "transparent". Of course, appropriate logical operations between bit maps can achieve the same effect [SALE86], but the programmer must consciously translate his or her objective of "transparent" into the appropriate boolean operation.

The other related work is that of Zachrisen [ZACH84], who describes a system for subdividing a frame buffer into multiple logically distinct images (called layers in the paper). The images are views superimposed one upon the other. Each image has a priority, with the front-most image having the highest priority; the rear-most, the lowest. Each image has its own logical look-up table, along with a single pixel value which represents transparent. No operators between images are provided, nor is the concept of an image frame provided. A recursive procedure for loading the physical lookup table is given. The algorithm begins with the highest-priority image, and recurs for each lower-priority image.

3. ICL Concepts

Central to ICL are images, which are the size of the frame buffer in height and width. Each image has a programmer-specified number of bits per pixel (k), and a logical look-up table with 2^k entries. Images can be created and deleted, and entries in the logical look-up table associated with each image can be set to either a color or to a special value of **transparent**. Pixels with this value are not displayed, and are treated in a special way discussed below. Colors are specified as {red, green, blue} triplets whose individual values are in the range of [0.0, 1.0]. Images have character-string names.

Composition expressions operate on one or more images, literals, and scalar program variables. Unlike RasterOp, which operates on pixel values, composition expressions operate on the color values of the pixels. There are two types of composition expressions: Simple, and If. Simple composition expressions are formed with the operators +, -, /, *, **min**, and **max**.

The composition expression

{1.0, 1.0, 1.0}

is a red, green, blue color triplet literal which evaluates to create a field of white.

Similarly,

{0.0, 0.0, 0.0}

evaluates to black, while

{1.0, 0.0, 0.0}

evaluates to red, etc. Black can be expressed more tersely as the scalar literal 0.0, white as 1.0, and grays as intermediate values.

Further examples of composition expressions are based on the simple test images *a* and *b*, shown in Figures 1 and 2. The result of the expression

$a + b$

is shown in Figure 3. The + operator is performed component by component on *a* and *b*, with the component sum defined to be either the actual sum or 1.0, whichever is less.

The difference of *a* and *b* is shown in Figure 4. The individual components of an ICL expression, once evaluated, are clamped from above at 1.0 and from below at 0.0. This is of course done because the RGB color space is defined with each component in the [0.0, 1.0] interval. Similar clamping would be needed with other color spaces. The clamping is not done after each operator is applied, but *only* after an entire expression is evaluated.

Figure 5 illustrates the **max** operator via the expression

a **max** *b*

ORIGINAL PAGE IS
OF POOR QUALITY

Figure 1. *ImageA*, used to illustrate composition expressions. The vertical stripes have intensities of 1.0, .75, .50, .25, and 0.0.

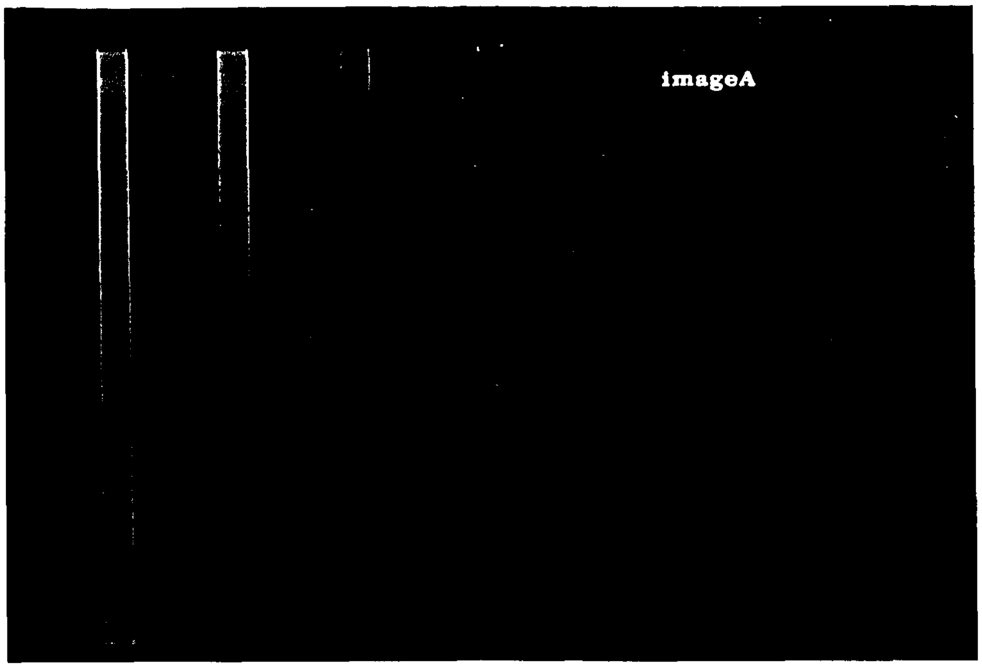
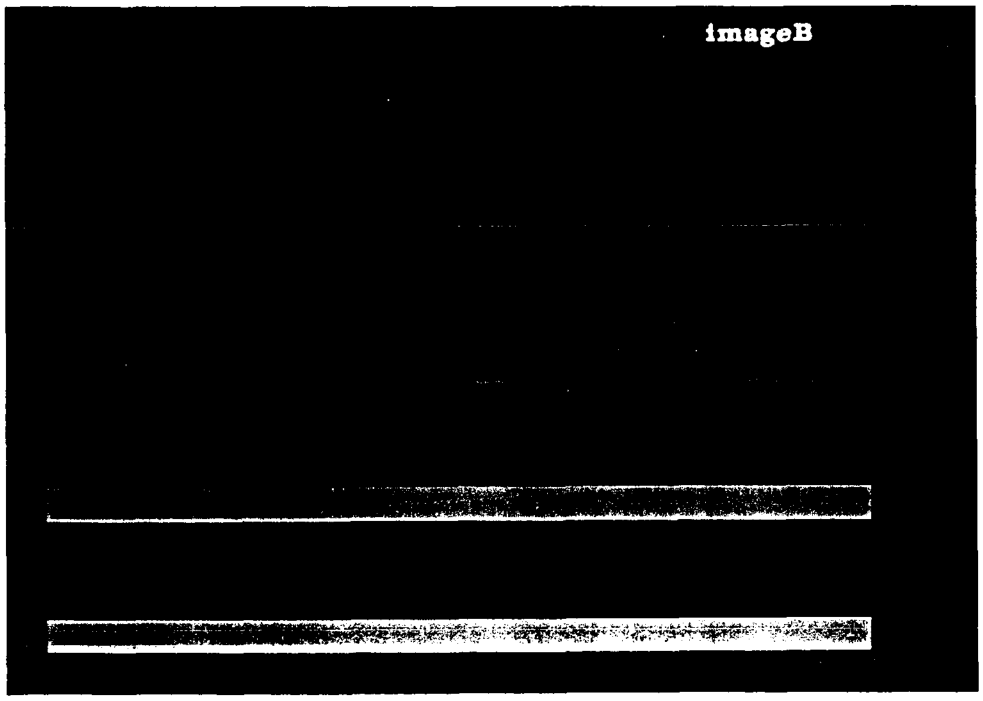


Figure 2. *ImageB*, used to illustrate composition expressions. The intensities are the same as in Figure 1.



ORIGINAL PAGE IS
OF POOR QUALITY

Figure 3. The sum of the two images, *ImageA+ ImageB.*

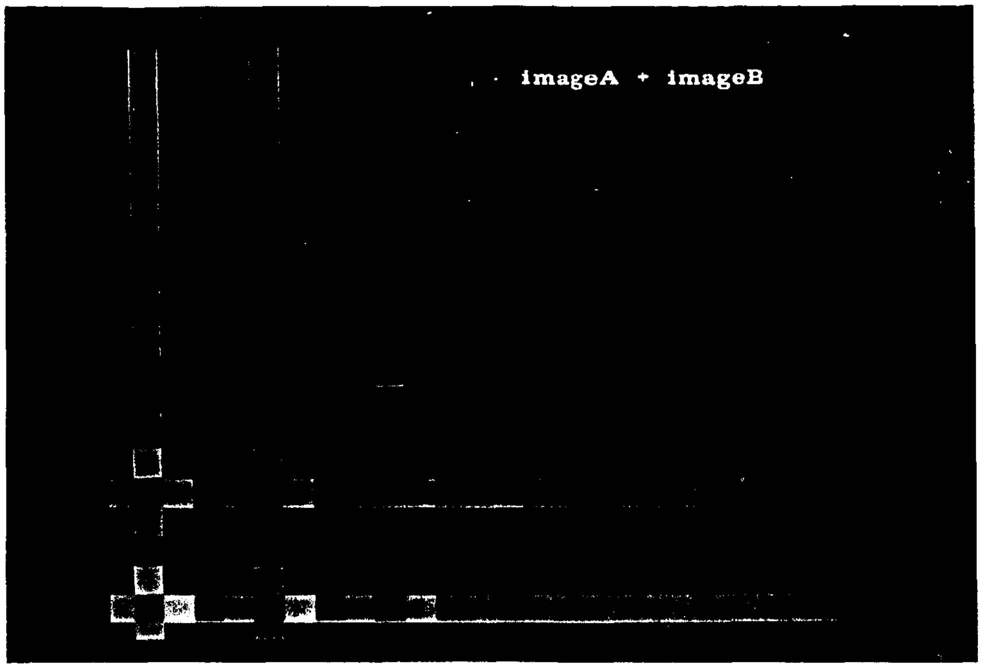
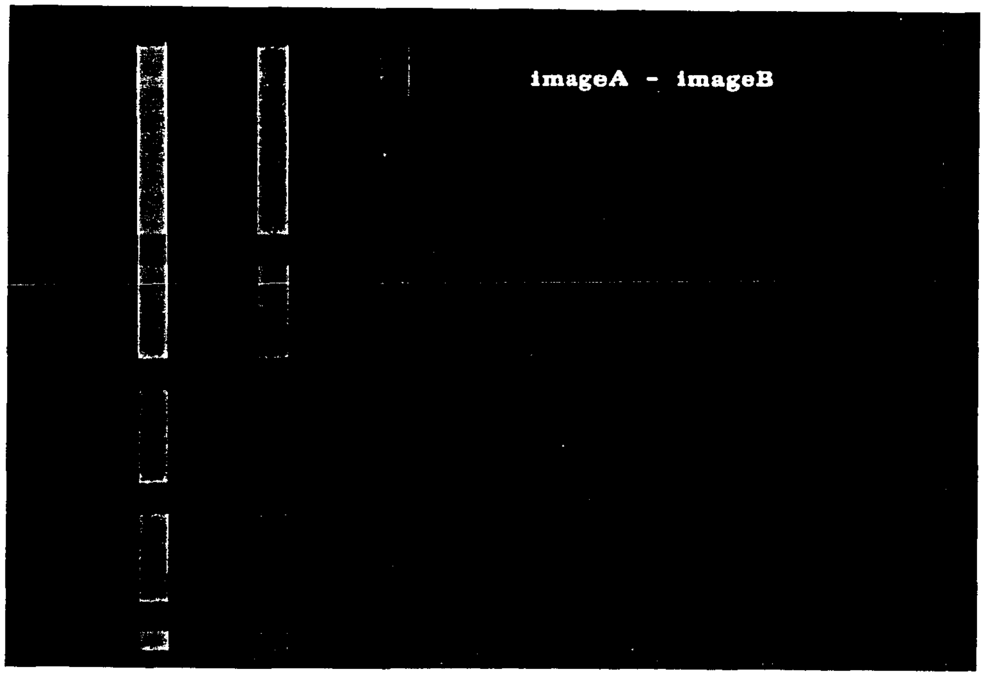


Figure 4. The difference of the two images, *ImageA- ImageB.*



ORIGINAL PAGE IS
OF POOR QUALITY

Figure 5. Images *a* and *b* combined with the **max** operator, *ImageA* **max** *ImageB*.

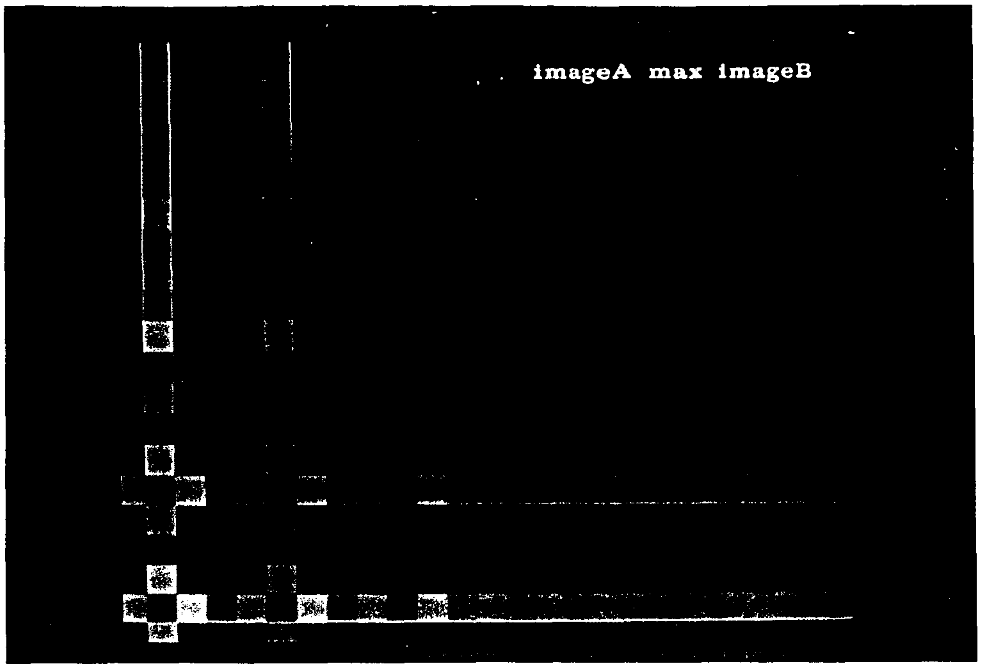


Figure 6. Images *a* and *b* combined with the **min** operator, *ImageA* **min** *ImageB*.

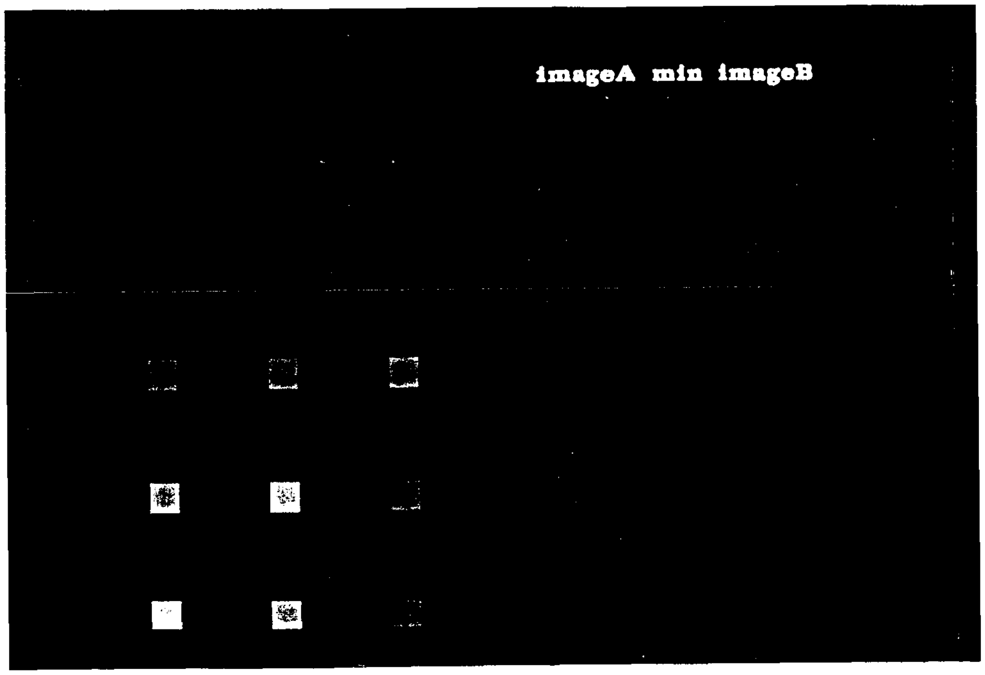


Figure 6 illustrates the **min** operator via the expression

```
    a min b
```

If *t* is a scalar variable, then the expression

```
    a * (t, t, t)
```

multiples the color of each pixel in *a* times *t*. Using the same convention for shortening literals which have equal components, this can also be written as:

```
    a * t
```

Given the image of *Mona _Lisa* shown in Figure 7, the expression:

```
    (Mona_Lisa  - 0.5) * 2
```

expands the range of each color component's values from .5 to 1.0 into 0.0 to 1.0, and maps all the values from 0.0 to 0.5 into the value 0.0, as shown in Figure 8. This type of color range expansion is common in image processing work.

Thus far we have discussed only simple composition expressions. The other type of composition expression is the conditional composition expression, which is of the form:

```
    if Conditional_Expression
        then Then_Composition_Expression
        else  Else_Composition_Expression
    endif
```

where the indentation is used solely for clarity of presentation. The Then and Else composition expressions can either be simple composition expressions or further conditional composition expressions. The Conditional_Expression is formed with relational operators (=, <>, <, <=, >, =>), and the results of the relational operators can be combined with the **and, or, and not** operators. The conditional composition expression takes on the value of either the Then_Composition_Expression or the Else_Composition_Expression, depending on whether the Conditional_Expression is **true** or **false**. If the Conditional_Expression evaluates to **false** and the optional **else** clause is missing, then the value is **transparent**. The effect of this value is explained below.

A simple conditional composition expression is:

```
    if Mona_Lisa  > 0.6 then 1.0 else 0.0  endif
```

which contours the gray level image *Mona Lisa* at the gray value of 0.6 as shown in Figure 9.

Another example, showing the nesting of the if statement, is:

```
    if Mona_Lisa >0.9 then {0.8, 0.8, 1.0}
        else if Mona_Lisa >0.7 then {0.6, 0.6, 0.8}
        else if Mona_Lisa >0.5 then {0.4, 0.4, 0.6}
        else if Mona_Lisa >0.3 then {0.2, 0.2, 0.4}
        else {0.1, 0.1, 0.2}
    endif endif endif endif
```

The result of applying this exxpression to the *Mona_Lisa* image is shown in Figure 10.

Further examples of ICL expression are based on the graytone image *c* of Figure 11. Figure 12 shows the use of a more complex condtional expression with nested if-then-else statements to replace specific gray values with colored lines. The expression is:

ORIGINAL PAGE IS
OF POOR QUALITY

Figure 7. The image *Mona_Lisa*, a digitized black and white halftone photo.

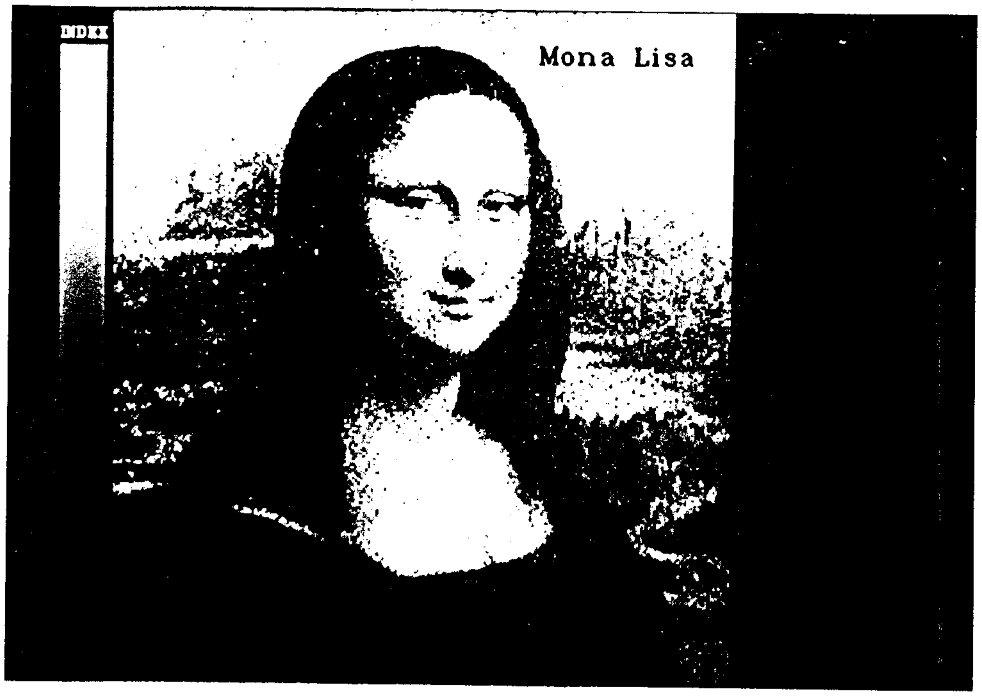
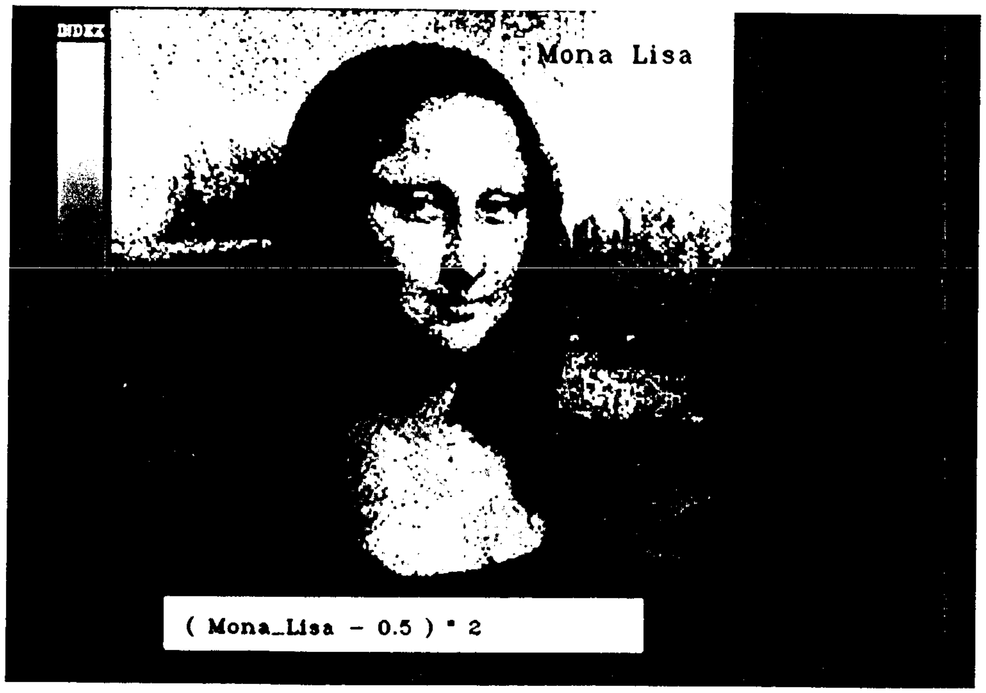
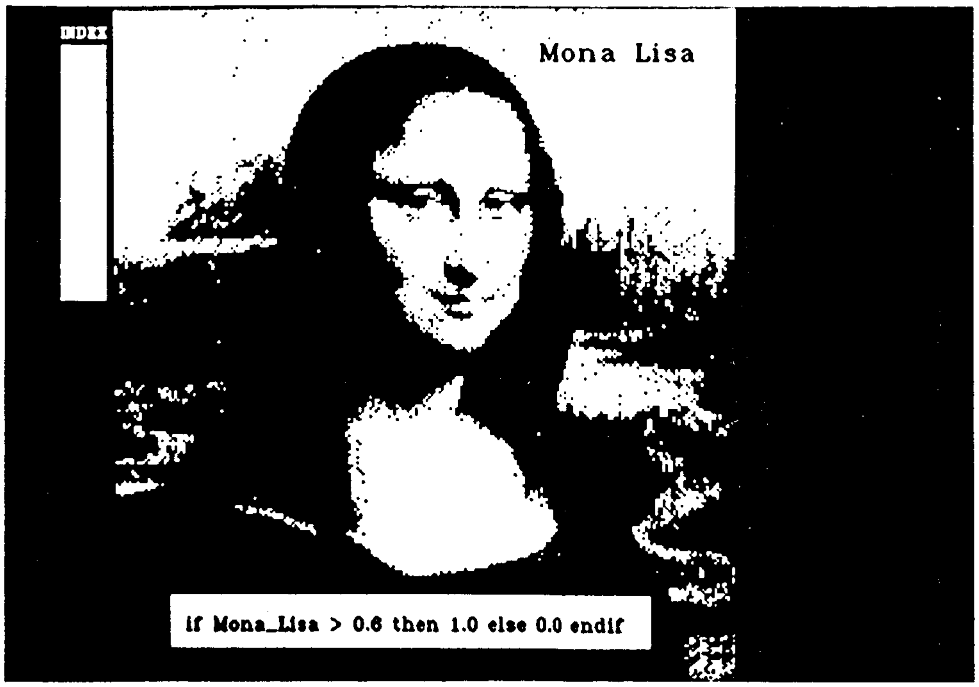


Figure 8. Expansion of the range [0.5, 1.0] into the range [0.0, 1,0], using the compostion expression (*Mona_Lisa* - 0.5) * 2.



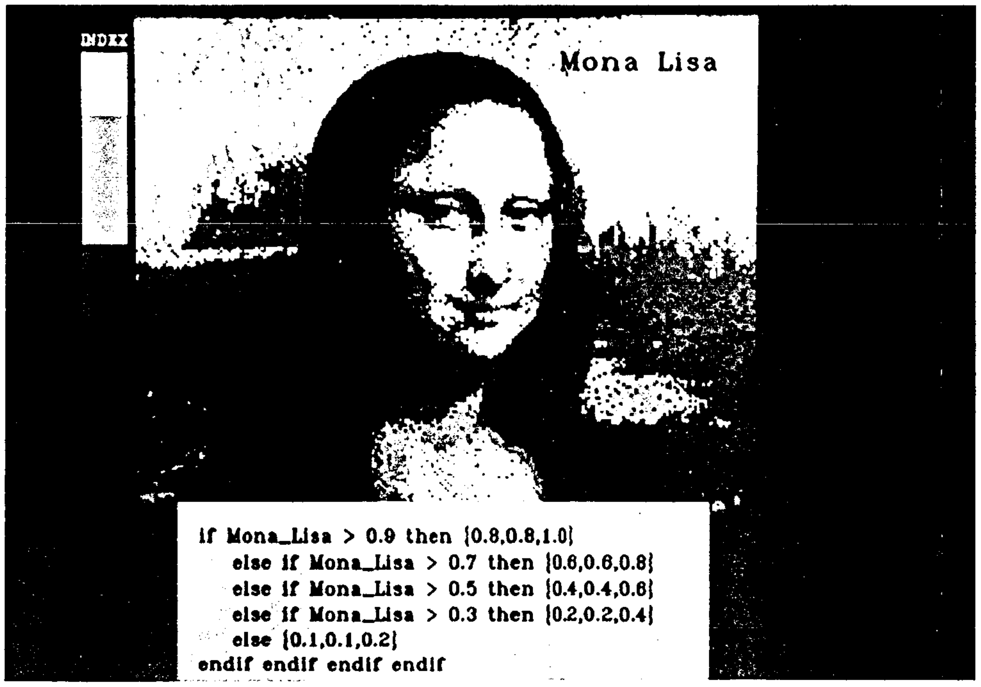
ORIGINAL PAGE IS
OF POOR QUALITY

Figure 9. The half-tone image *Mona_Lisa* contoured at an intensity level of .6, so that all values greater than .6 are white, and the rest are black. The composition expression is: **if** *Mona_LIsa* > 0.6 **then** 1.0 **else** 0.0 **endif**.



```
if Mona_Lisa > 0.6 then 1.0 else 0.0 endif
```

Figure 10. The *Mona Lisa* image modified using a nested **if ... then ... else** statement.



```
if Mona_Lisa > 0.9 then {0.8,0.8,1.0}
   else if Mona_Lisa > 0.7 then {0.6,0.6,0.8}
   else if Mona_Lisa > 0.5 then {0.4,0.4,0.6}
   else if Mona_Lisa > 0.3 then {0.2,0.2,0.4}
   else {0.1,0.1,0.2}
endif endif endif endif
```

ORIGINAL PAGE IS
OF POOR QUALITY

Figure 11. A graytone image *c*.

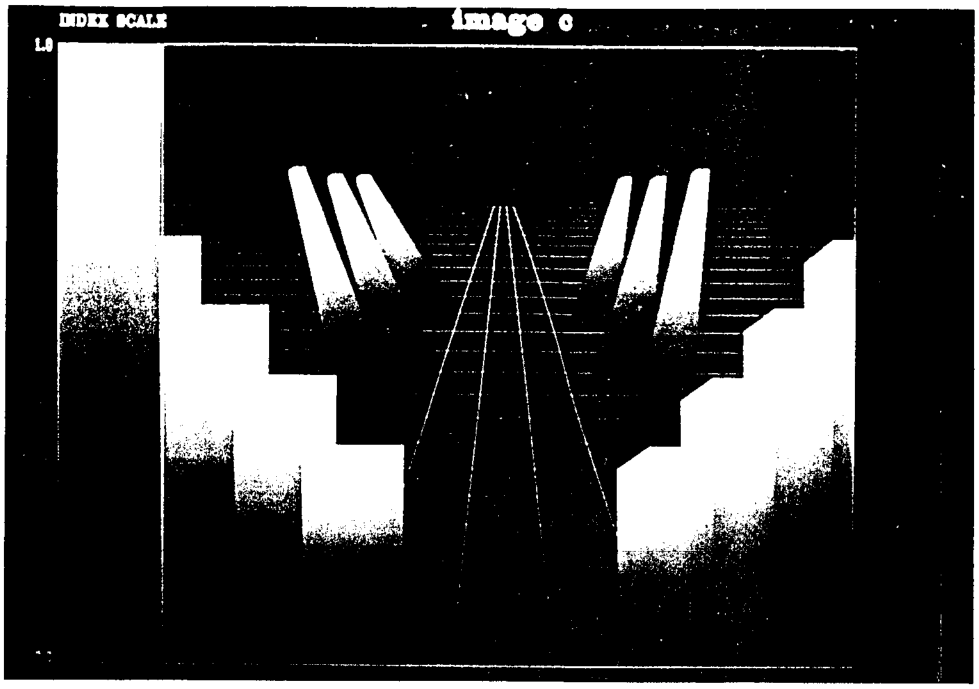
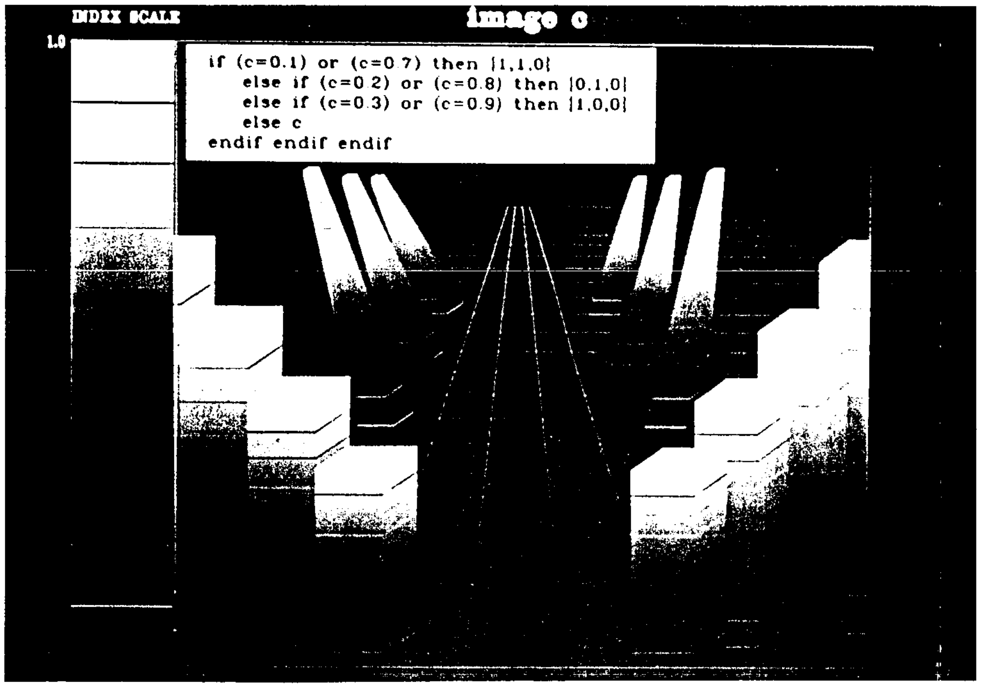


Figure 12. The result of applying a nested **if ... then ... else** statement with **or** used in the conditional expression.



```
if (c=0.1) or (c=0.7) then {1,1,0}
    else if (c=0.2) or (c=0.8) then {0,1,0}
    else if (c=0.3) or (c=0.9) then {1,0,0}
    else c
endif endif endif
```

```
if (c=0.1) or (c=0.7) then {1,1,0}
   else if (c=0.2) or (c=0.8) then {0,1,0}
   else if (c=0.3) or (c=0.9) then {1,0,0}
   else c
endif endif endif
```

In Figure 13 we see the results of a similar expression, but with ranges of intensities being modified instead of specific values:

```
if (c<0.8) and (c>0.6) then c+{1,0,0}
   else if (c<0.5) and (c>0.3) then c+{0,1,0}
   else c
endif endif
```

Figure 14 shows text with filled interiors, and Figure 15 shows the same text with empty interior. This effect is obtained by subtracting the interior color using the following expression:

```
text - {1,0,0}
```

Selectors can be used to designate the red, green, or blue component of an image used in a Conditional_Expression:

```
if a.r > 0.5 then a - {0.5, 0, 0} else a endif
```

This expression removes some red from all those pixels in *a* which have more than 0.5 red, and leaves the other pixels unchanged.

The diagrams in Appendix A give precise definitions for the syntax of compositon expressions.

The LUTC is implemented as a subroutine package: ICL expressions are passed as character strings at run time, then parsed and evaluated. Because ICL is currently implemented as a subroutine package rather than as a language extension, image and scalar variables are declared by procedure calls.

In the following paragraphs, we discuss some of the subroutine calls which make up ICL. A complete list of the calls is in Appendix B. Type definitions assumed by the subroutine calls are as follows:

```
type
        PixelRange  = 1..9;
        IndexRange  = 0..511;
        ColorRange  = real;
        Rectangle   = record
                            xmin, xmax, ymin, ymax :  cardinal;
                      end;
```

Images are defined with the call:

```
        DefineImage          (ImageName       : array of char    ;
                              BitsPerPixel    : PixelRange       ),
```

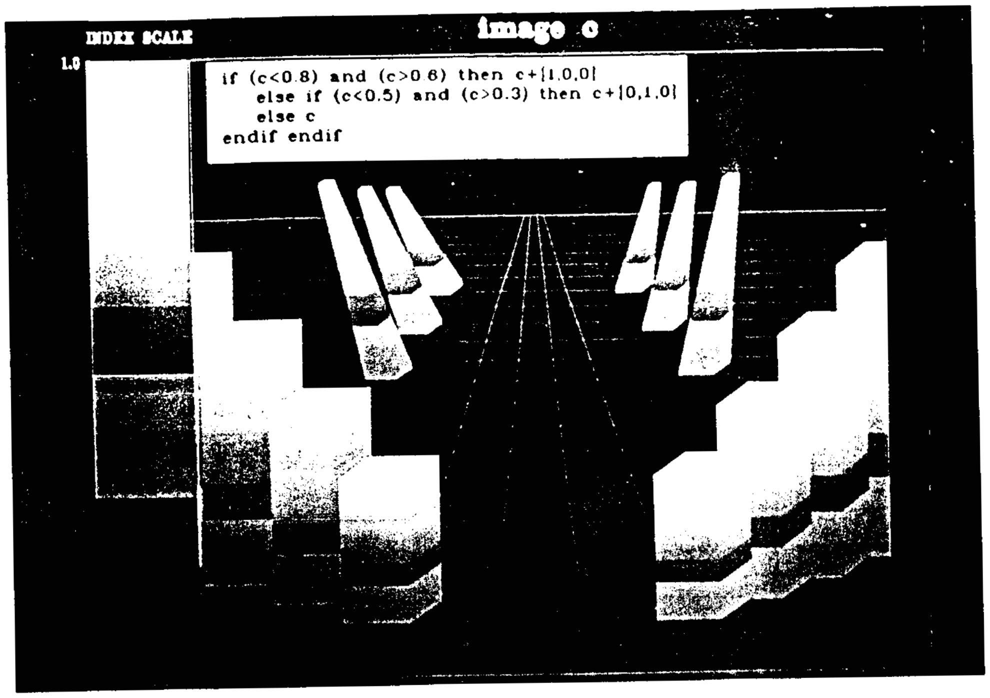
which allocates the requisite number of planes in the refresh buffer to the image.

Scalar variables used in ICL expressions are actual Modula-2 program variables. They are declared to LUTC by:

```
        DeclareVar           (VariableName    : array of char    ;
```

ORIGINAL PAGE IS
OF POOR QUALITY

Figure 13. The result of applying a nested **if ... then ... else** statement with **and** used in the conditional expression to specify ranges of values.



```
if (c<0.8) and (c>0.6) then c+[1,0,0]
   else if (c<0.5) and (c>0.3) then c+[0,1,0]
   else c
endif endif
```

ORIGINAL PAGE IS
OF POOR QUALITY

Figure 14. Text strings with filled interior.

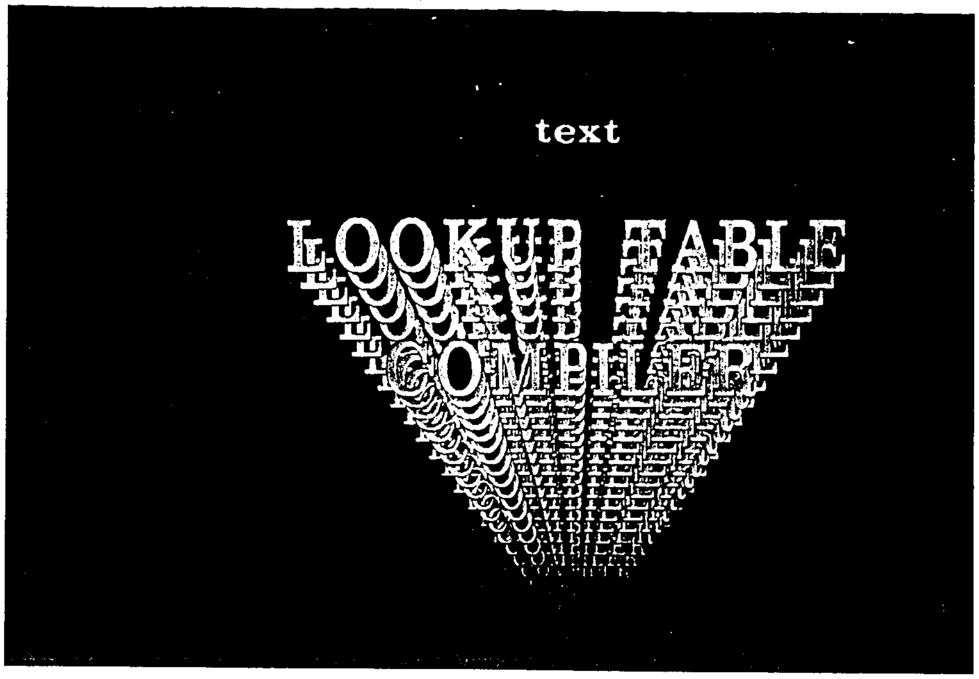
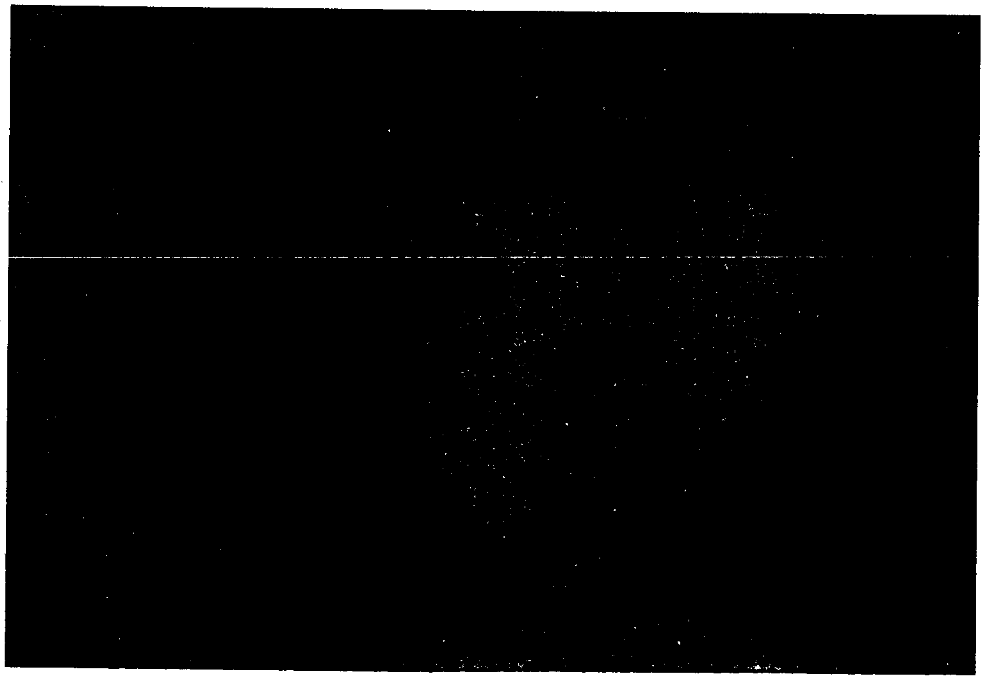


Figure 15. Text strings with empty interior, created by subtracting the interior color from the original image.



```
                    VariableAddress     : address           ),
```

where *VariableAddress*, as the name implies, is the address of the variable. This is used at run time to obtain the value of variables used in composition expressions. Scalar variables must be of type **real**.

The color associated with the pixel value of a particular image is set with the procedure:

```
SetImageIndexColor      (ImageName          : array of char    ;
                         Index              : IndexRange       ;
                         red,green,blue     : ColorRange       );
```

The procedure call:

```
DefineCompositionFrame  (FrameName          : array of char    ;
                         Bounds             : Rectangle        ;
                         CompositionExpression  : array of char),
```

defines to the LUTC both a rectangular composition frame and an associated composition expression. The expression is to be applied within the composition frame. Multiple composition frames and their accompanying composition expressions can be defined by successive calls to **DefineCompositionFrame**. The entire set of defined frames are displayed via a call to the procedure **DisplayCompositionImage**. If several frames overlap, then temporal priority rules: the most recently defined frame has the highest priority.

Composition frames are not restricted to be a rectangle. They can be modified by the procedure calls:

```
ExtendCompositionFrame  (FrameName          : array of char    ;
                         Bounds             : Rectangle        );
ReduceCompositionFrame  (FrameName          : array of char    ;
                         Bounds             : Rectangle        );
```

A complete list of LUTC procedure calls is given in Appendix B.

4. Transparent Pixels

One concept in ICL is worthy of separate discussion - transparent pixels. Any number of pixel values associated with an image can be declared to be transparent by calling the procedure:

```
SetImageIndexTransparency (Image Name       : array of char    ;
                           Index            : IndexRange       ;
                           Transparency     : boolean          );
```

The underlying motivation for transparent pixels is that images being composed together are often cropped along irregular paths, such as a picture of a ship being cropped along the silhouette edge of the ship. Because the picture must be stored in an image, which is a rectangular array of pixel values, some way is needed to indicate which portions of the image are to enter into composition expression calculations, and which are to be ignored. For example, consider the array of pixel values shown in Figure 16. Pixel value 0 is used to designate those portions of the image which are not of interest, while values 1, 2, and 3 are actual values. To compose this image, which is called *picture*, with another image, called *background*, the following composition expression would be used:

if *picture* <> **transparent then** *picture* **else** *background* **endif**

0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	3	3	0	0	0
0	0	0	0	0	1	1	3	3	0	0	0
0	0	0	0	1	1	1	1	3	0	0	0
0	0	0	1	1	1	1	1	1	0	0	0
0	0	1	1	1	1	1	1	1	1	0	0
0	0	1	1	2	2	2	1	1	1	0	0
0	0	1	1	2	2	2	1	1	1	0	0
0	0	1	1	2	2	2	1	1	1	0	0
0	0	1	1	2	2	2	1	1	1	0	0
0	0	1	1	1	1	1	1	1	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0

Figure 16. A two-bit per pixel image, with the pixel value of 0 corresponding to the background value, which is treated as Transparent. Pixel values 1, 2, and 3 are part of the actual image.

ICL might have been designed differently, such that this effect would be achieved in some other way. First, a mask, consisting of a one-bit per pixel image, might be used to define which part of PICT is to be used in the composition, as in the recent "Two-Bit Graphics" algebra for manipulating one-bit images accompanied by a one-bit transparency mask [SALE86], or as in [WARN82]. This was rejected because bit planes are scarce in a refresh buffer.

Alternatively, transparent pixels might have been rejected in favor of allowing pixel index values (rather than colors) to be used in the conditional, for instance:

> **if** *picture.index* <> 0 **then** *picture* **else** *background* **endif**

While viable, this was rejected because we wanted to limit ICL statements to image colors.

Another alternative would be for the programmer to assign some color, say {.7, .7, .7}, to pixel index 0 (any other color would be equally suitable). Then the IF statement would be:

> **if** *picture* <> {.7, .7, .7} **then** *picture* **else** *background* **endif**

This was rejected for two reasons. First, the use of a value from a range of values to designate something different than the other values is poor software engineering practice (akin to a function returning a zero if the function could not be evaluated for the given argument, otherwise returning the correct value. Second, what if {.7, .7, .7} also just happens to be the actual color of some pixel value in the image? There is no way to determine if a particular pixel with this value is to be treated as transparent, or as a pixel to be displayed.

Yet another possibility is to define geometric outlines, or "key holes", through which an image could be viewed, such as the shape stencils of [WARN82], the mask regions of Quickdraw [APPL85], or the clipping paths of PostScript [ADOB85]. We rejected this for the sake of simplicity.

Arithmetic operations between transparent pixels return **transparent** as a value. For operations between one transparent and one regular pixel value, **transparent** is defined as the identity element for the operation in question: as 0 for +, -, and **max**; as 1 for *, /, and **min**. This means that transparent pixels essentially do not enter into compositions. For relational operators, a comparison between two transparent pixels is **true** for =, <=, and >=, and **false** for the other comparisons. Comparisons between one transparent pixel and one non-transparent pixel return **false** except for <>, which is **true**. The **false** results are because there is no ordering between the value **transparent** and the color triplets.

6. Further Examples

In this section we further illustrate ICL concepts. To add together three images stored in the refresh buffer, the expression:

$$a/3 + b/3 + c/3$$

or the expression:

$$(a + b + c)/3$$

can be used: the expressions are equivalent, because the range of intermediate results is not limited to the [0.0, 1.0] interval (as discussed in Section 3).

Imagine three monochrome images called *triangle*, *rectangle*, and *circle* with pixel value 0 (the background value) having been declared to have value **transparent**. To display the images such that *triangle* is displayed on top of *rectangle* which is in turn displayed on top of *circle*, we use the expression:

```
if triangle <> transparent then {0,0,1}
   else if rectangle <> transparent then {0,1,0}
   else if circle <> transparent then {1,0,0}
   else 0.0
endif endif endif
```

Figure 17 illustrates this prioritization effect, which is further discussed in [FOLE82, page 491].

Many operations required by window managers can be implemented in ICL. If the images for the two windows are *win1* and *win2*, then the sequence of calls:

```
DefineCompositionFrame  ("one", window1_boundary,"win1")
DefineCompositionFrame  ("two", window2_boundary,"win2")
DisplayComposedImage
```

will display *win2* on top of *win1*, because the temporal priority of composition frame definition determines how to resolve the visibility conflict when composition frames overlap. The variables *window1_boundary* and *window2_boundary* are assumed to be of type rectangle, as previously defined. Resizing window 1, while maintaining its visibility with respect to window 2, would simply involve using the calls:

```
ReduceCompositionFrame  ("one", window1_boundary)
{Assign new boundary values to the record window1_boundary}
ExtendCompositionFrame  ("one", window1_boundary)
```

Reversing the visibility order of then two windows so that *win1* is on top involves just:

```
DefineCompositionFrame  ("two", window2_boundary,"win2")
DefineCompositionFrame  ("one", window1_boundary,"win1")
DisplayComposedImage
```

In this example, the composition expressions associated with composition frames one and two are the single images *win1* and *win2*.

More complicated images can be created, as in Figure 18 which shows the results of applying composition expressions to the image of *Mona_Lisa*. Frame *one* is defined as *whole_area* (0,1023,0,1023), reduced by the small *upper_right_area* (520,820,500,800) and *vertical_strip* (500,510,0,1023) using **ReduceCompositionFrame**. The frame on top named *two* is defined after frame *one*, and consists of two disjoint areas, *right_half* (510,1023,0,1023) and *lower_left_area* (180,480,200,500), created with **ExtendCompositionFrame**:

```
DefineCompositionFrame  ("one",whole_area,"Mona_Lisa+{0,0,0.3}")
ReduceCompositionFrame  ("one",upper_right_area)
ReduceCompositionFrame  ("one",vertical_strip)
DefineCompositionFrame  ("two",right_half,"Mona_Lisa+{1,0,0}")
ExtendCompositionFrame  ("two",lower_left_area)
DisplayComposedImage
```

ORIGINAL PAGE IS
OF POOR QUALITY

Figure 17. The effect of prioritizing three monochrome (one-bit) images on top of one another with a nested **if ... then ... else** statement.

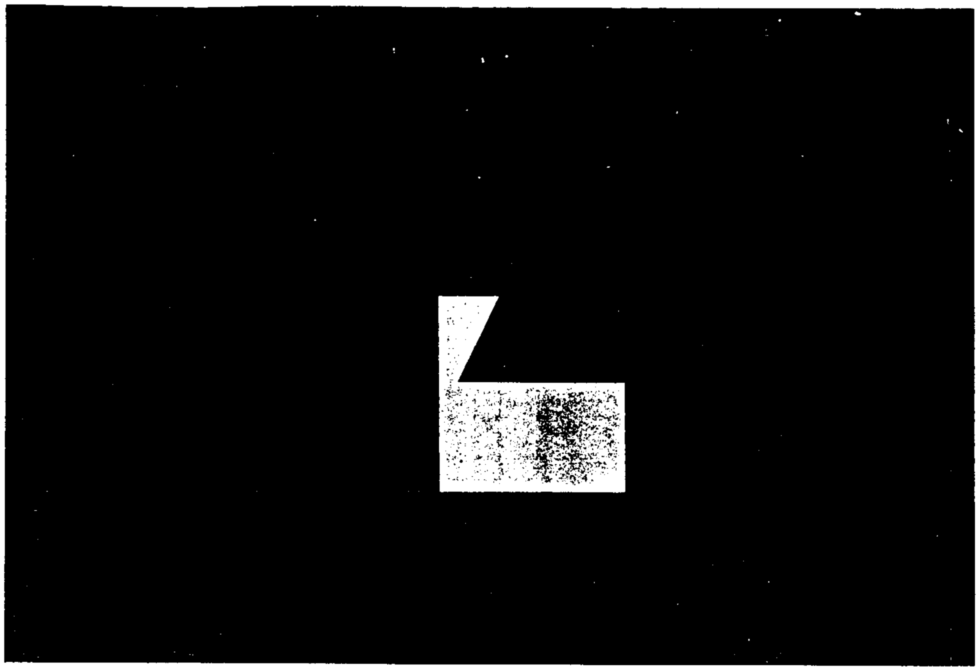
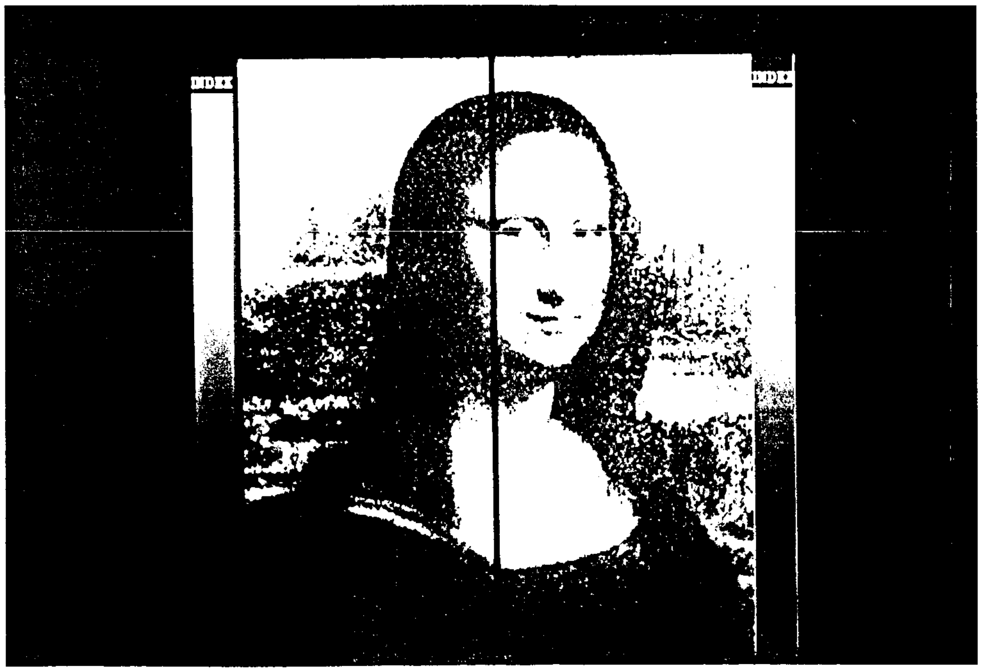


Figure 18. Two overlapping frames with different composition expressions. The frames were created with the **extend** and **reduce** procedures.



Another application of ICL is to produce 'lap dissolve' images, in which one image slowly fades out and is replaced by another image. The basic idea is that a composition expression:

*imageA**(1 - *t*) + *t***imageB*

is continually applied as the parameter value t varies from 0 to 1. This example shows the advantage of including scalar variables in composition expressions: changing the effect of the expression simply involves changing the value of the variable(s) used in the expression, and reevaluating. The code to produce such an effect is simply:

```
DeclareVar   ("t",ADR(t));                  {tell LUTC about t           }
DefineImage ("imageA", 5);                  {an image with 5 bits/pixel }
DefineImage ("imageB", 5);                  {an image with 5 bits/pixel }
DefineCompositionFrame
            ("LapDissolve", boundary, "imageA*(1-t) + imageB*t ");
for i := 0 to n do                          {n is number of steps        }
   t := FLOAT(i)/FLOAT(n);
   DisplayComposedImage;
   {might want to introduce a time delay here}
end ;
```

Several steps in the lap-dissolve sequence, with values of *t* = 0.0, 0.2, 0.4, 0.6, 0.8, and 1.0, are shown in Figures 19.1 -19.6.

A very easy way to allow a user to control the red/green/blue color balance and overall brightness of an image is with the expression:

(*image* *{*red*, 0, 0} + *image* *{0, *green*, 0} + *image* * {0, 0, *blue*}) * *brightness.*

The scalar variables *red, green, blue,* and *brightness* would all be set with slider dials or some similar dynamic interaction technique.

In summary, ICL can be used in a variety of ways to combine multiple images stored in the refresh buffer.

6. Implementation

The LUTC has been implemented on a VAX 11/780, under VMS, using a 10-plane RAMTEK 9400. The LUTC is a run-time subroutine package. The implementation language is MODULA-2. The main reason for the choice of this language is its ability to pass the addresses of variables declared in the program, in order to build our variable table. PASCAL allows only dynamic creation of pointers. MODULA-2 does not impose this restriction; the addresses of declared variables within the program may be obtained, using the low-level system facilities. With this capability, composition expressions can include numeric variables. Other reasons for the choice of MODULA-2 as the implementation language include its modularity, separate compilation capability, and high-level data structure facility.

There are two main modules: the first builds a symbol table, based on image and variable declarations, and the other creates composition frames and evaluates composition expressions. These and other modules are shown in Figure 20.

Figure 19.1. A lap-dissolve sequence, showing the effects of the parameter *t* at 0.0.

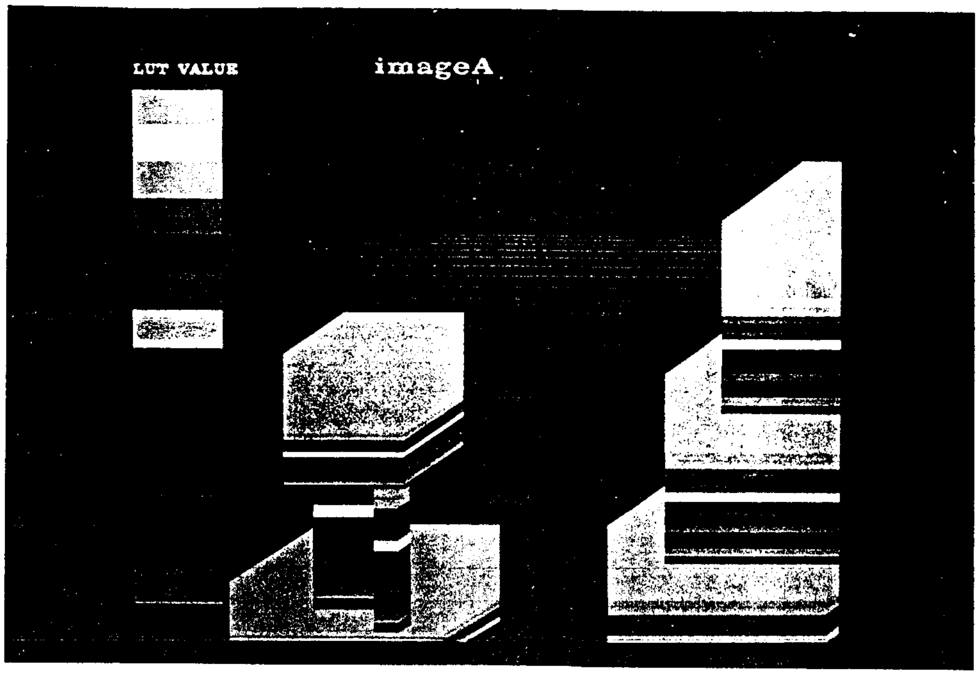
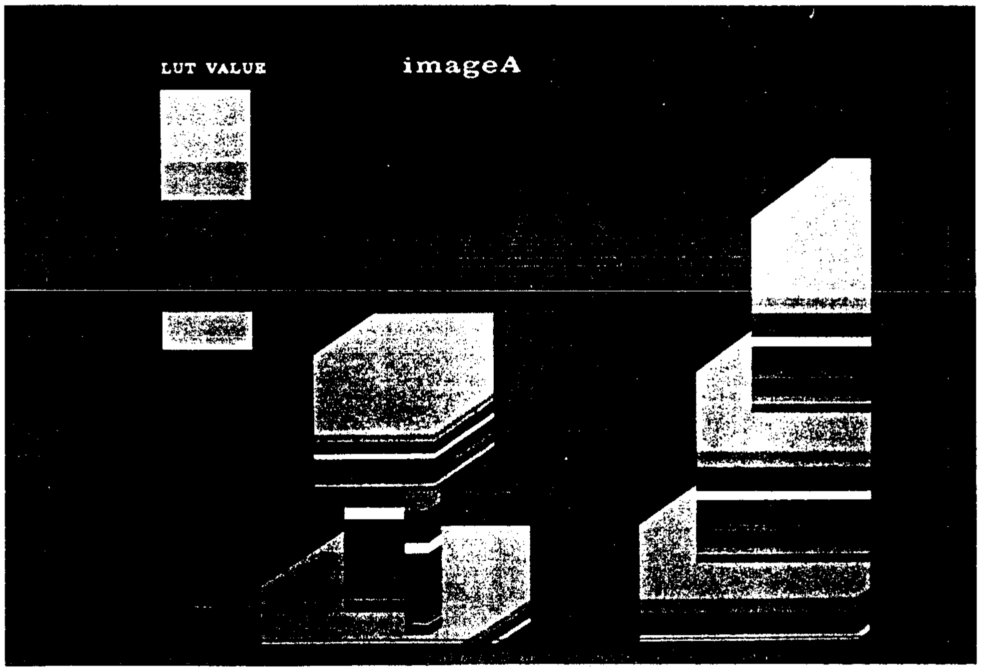


Figure 19.2. A lap-dissolve sequence, showing the effects of the parameter *t* at 0.2.



ORIGINAL PAGE IS
OF POOR QUALITY

Figure 19.3. A lap-dissolve sequence, showing the effects of the parameter *t* at 0.4.

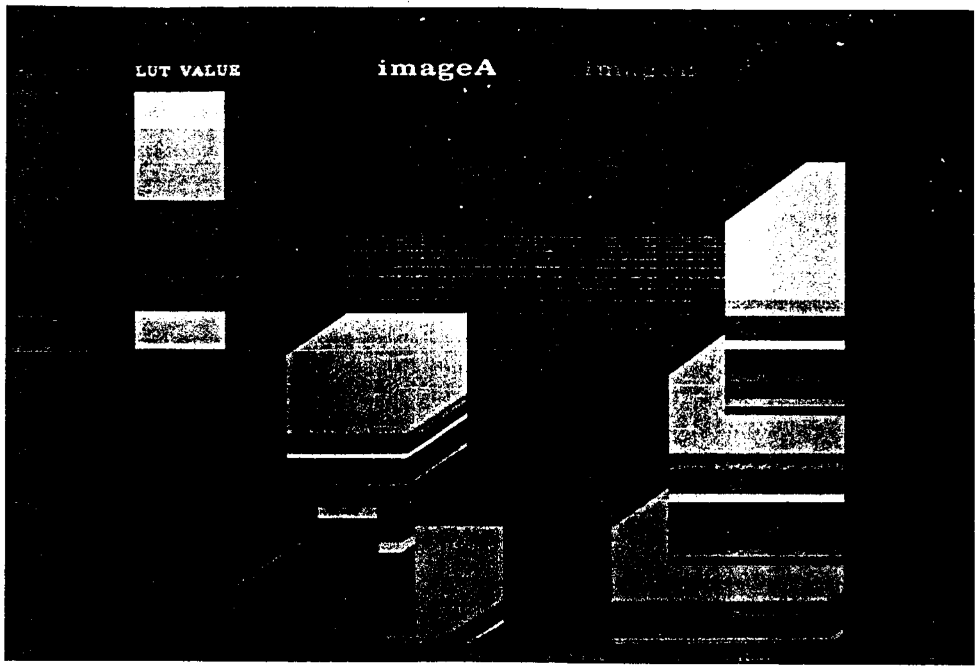
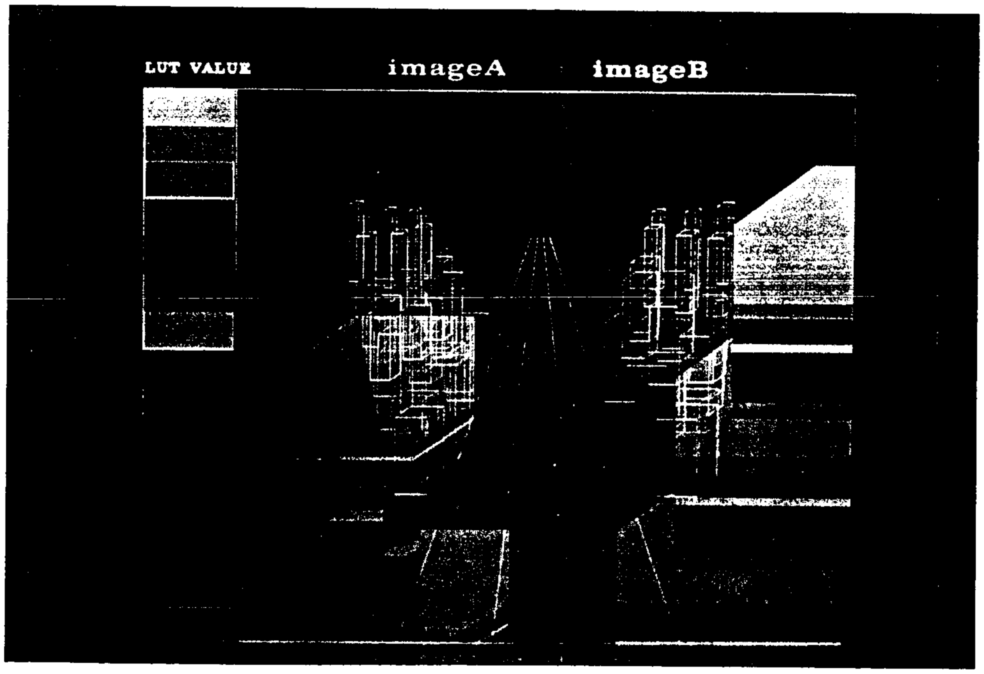


Figure 19.4. A lap-dissolve sequence, showing the effects of the parameter *t* at 0.6.



ORIGINAL PAGE IS
OF POOR QUALITY

Figure 19.5. A lap-dissolve sequence, showing the effects of the parameter *t* at 0.8.

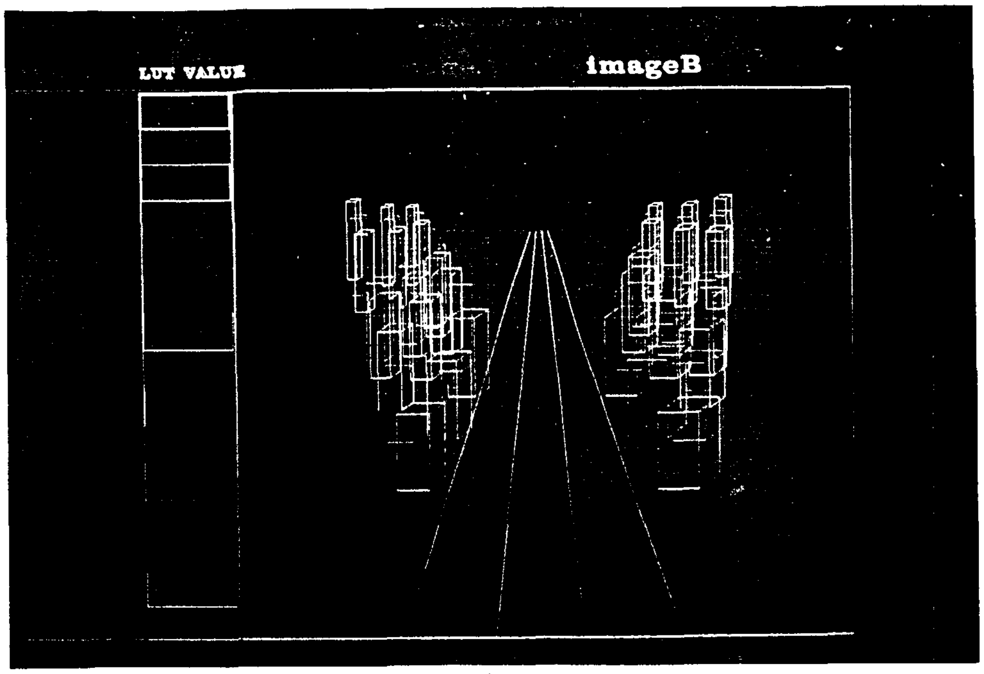
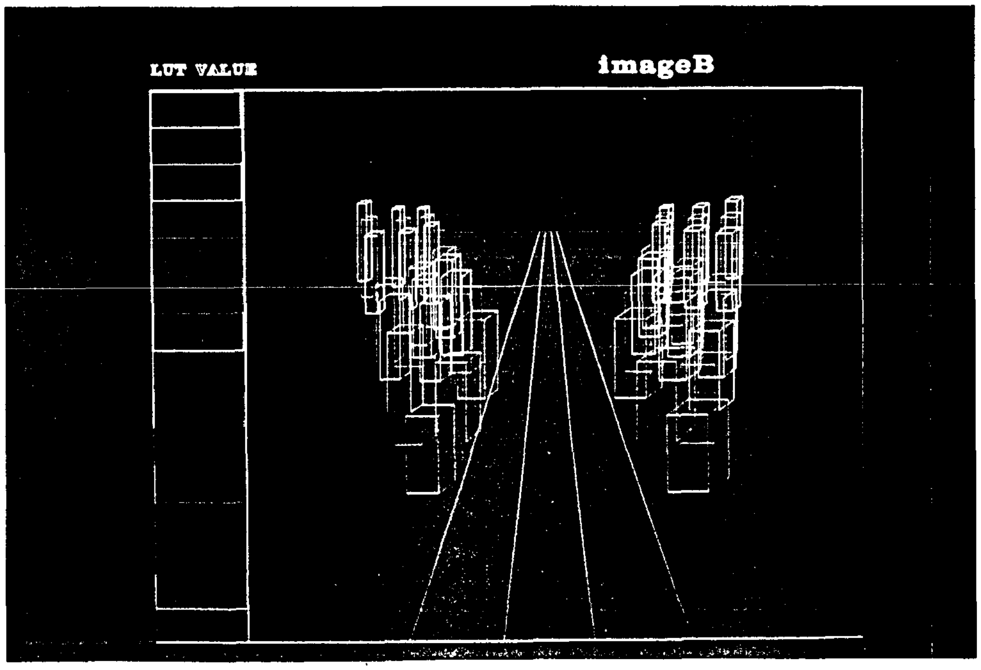


Figure 19.6. A lap-dissolve sequence, showing the effects of the parameter *t* at 1.0.



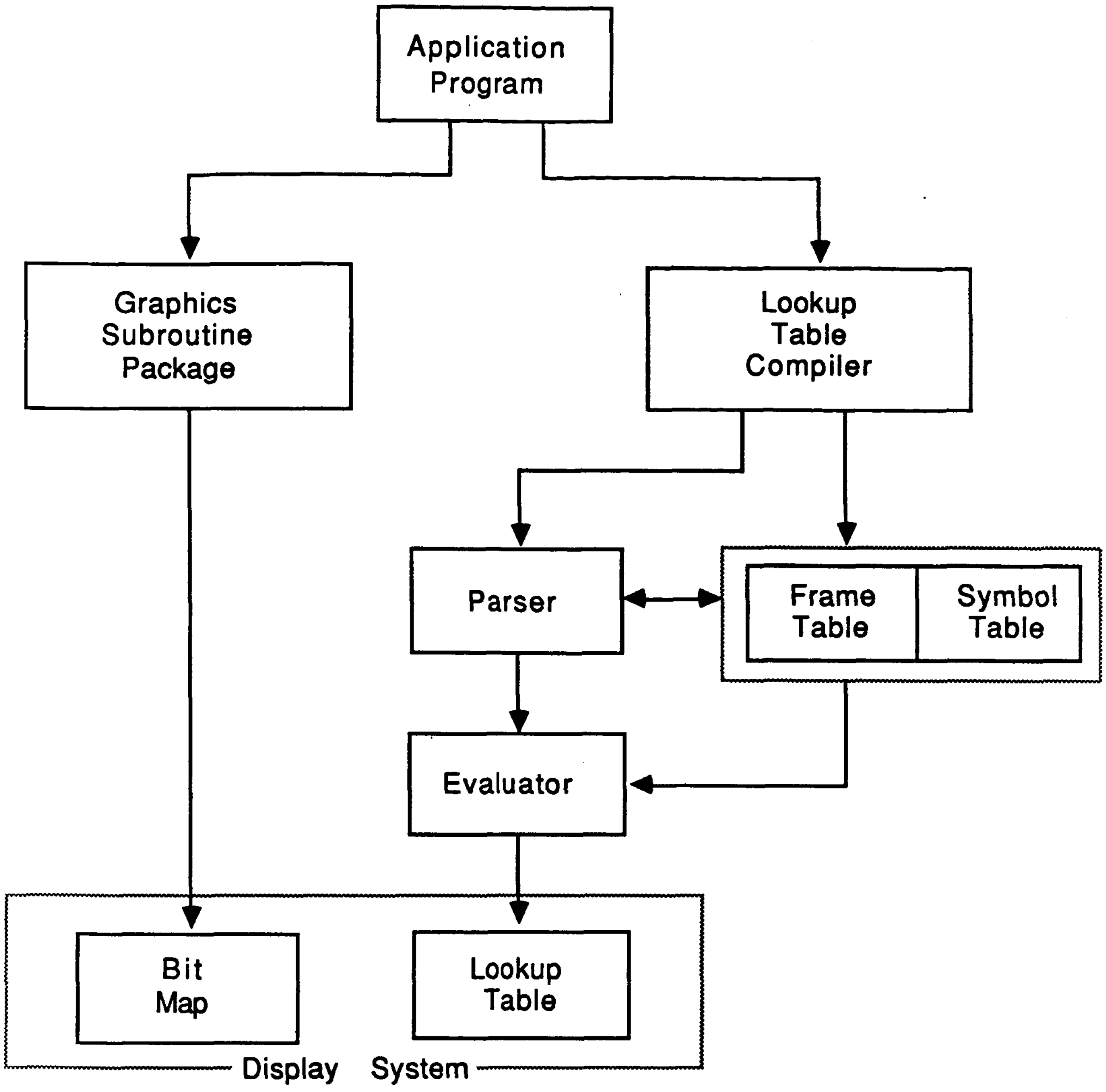


Figure 20. An overview of the LUTC and how it is integrated with the application program and display system. The application program calls to the graphics subroutine package cause images to be created in selected planes of the bit map. The LUTC loads the lookup table to control the appearance of the images.

The symbol table records the image name, bits per pixel, logical look up table and contents thereof (including tranparency of any pixel values), and a mask indicating which planes have been allocated to the image.

A composition frame record in then frame table contains its name and a mask indicating which plane has been allocated to the frame. This mask is used in evaluating a composition expression, to see if a particular look-up table value might be affected by the composition expression associated with the frame. The plane allocated to the composition frame contains the boundary of the frame as a one-bit mask. When a composition frame is defined, the plane is initialized to 0, and the frame rectangle is scan-converted into the plane as 1's. If the frame is extended, the new rectangle is added to the plane as 1's. If the frame is reduced, the reduction rectangle is scan converted into the frame as 0's.

When a composition expression is passed to the LUTC, it is parsed into a sequence of symbols and lexical tokens which are recognized by the composition language. Errors are also detected and flagged so that the expression is not used if **ComposeExpression** is called. The composition expression is tokenized because the expression will be repeatedly evaluated to actually load the look-up table, so it needs to be expressed in a readily accessable form rather than as a character string. The parser converts the infix expressions into postfix form and stores it in the composition frame record.

The parser handles use of parenthesis, nesting of **if-then-else-endif** statements, and precedence rules for the expression operators. The descending order is:

(*, /)
(+, -)
(min, max)
(>, >=, <, <=, =, <>)
(not)
(and, or)

The evaluator actually evaluates the tokenized postfix expressions and loads the look-up table. The evaluation uses a partial result stack. Each token is examined. If it is an operand, then the value is fetched and pushed onto the partial result stack. The value is not the pixel value of the image in question, but the value referenced by the pixel value in the logical lookup table. If it is an operator, then its operands are popped from the stack, the operator is applied, and the result is pushed back on the stack.

The evaluator does this for each lookup table entry, because the values for each image are different for each entry. As the evaluator cycles through each lookup table entry, it:

1. Finds the highest priority composition frame that affects this lookup table index. Recall that priority for overlapping composition frames is temporal.
2. Evaluates the highest priority composition expression for this index, clamping results to [0.0, 1.0].
3. Loads the lookup table with the value of the evaluated composition expression.

The time required to load the lookup table depends on the complexity of the expressions. For instance, parsing a simple binary expression (i.e. operand operator operand) takes less than .01 second of VAX 11/780 time, while creating the 1024-entry lookup table takes about one-half a second, or 500 µsec per entry. A three-deep **if** ... **if** ... **if** ... structure (i.e. **if** operand rel_op operand **then** result **else if** operand rel_op operand **then** result **else if** operand rel_op operand **then** result **else** result **endif endif**

endif) parses in .12 seconds, and requires two seconds for table generation. No attempt has yet been made to speed up these times.

7. Summary and Directions for Future Work

We have found the Image Composition Language and its associated LUTC to be a useful tool for increasing programmer productivity. A number of enhancements would make ICL be even more useful. Integrating composition expressions with the base language, MODULA, via a preprocessor would make programs more readable than at present. Implementing **DeleteImage** would provide more flexibility. Also useful would be language constructs for distinguishing between a pixel value and the lookup table value referenced by that pixel. This would allow classical raster operations to be made available through ICL.

Our colleague L. Henry has partially developed an alternative algorithmic approach to computing the look-up table values, which we would like to further explore. It involves computing an intermediate lookup table for each operator, and then using the intermediate lookup table to compute the next lookup table. For instance, in the composition expression *a + b/c*, the *b/c* expression would be evaluated to produce a lookup table, which we will call *temp*. Then the expression *a + temp* would be evaluated, producing the final lookup table. This could be more efficient than the present algorithm.

ICL can also be implemented with images of different sizes, with the results of the expression creating another image rather than just a view of the display. That is, ICL could be the basis for a fuller language for manipulating arbitrary bit map images. Such an implementation would want to take advantage of raster operation hardware found on many graphics-oriented workstations

Acknowledgements: We are pleased to acknowledge James Acquah for some of the ideas found in ICL and Chuck McMath for implementing an earlier version of ICL. The Graphics and User Interface Research Group at George Washington University provided an intellectual environment supporting this work, while financial support was provided by the Applied Mathematical Sciences Program, Office of Basic Energy Sciences of DOE, grant DE-AS05-83ER13122, and NASA-Langley Research Center, grant NAG-1-538-632200.

8. References

ACQU82 Acquah, J., J. Foley, J. Sibert, and P. Wenner, "A Conceptual Model of Raster Graphics Systems," SIGGRAPH '82 Proceedings, published as *Computer Graphics*, 16(3), August 1982.

ACQU84 Acquah, J., J. Foley, and C. McMath, *Graphics Programmings Language Research Status Report*, Report GWU-IIST-84-49, Dept. of EE&CS, George Washington University, Washington, D.C. 1984.

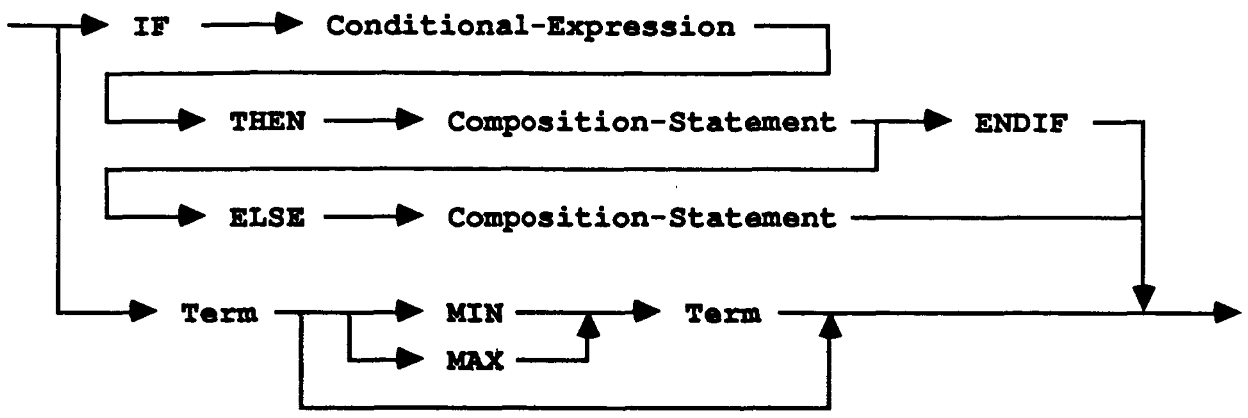
ADOB85 Adobe Systems, *PostScript Language Tutorial and Cookbook*, Addison-Wesley, Reading MA, 1985.

APPL85 Apple Computer, Inc., *Inside Macintosh*, Cupertino, CA, 1985.

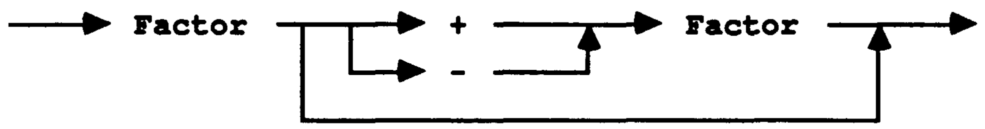
GARR82	Garrett, M. and J. Foley, "Graphics Programming Using a Database System with Dependency Declarations",*Transaction on Graphics*, 1(2), April 1982, pp. 109-128.
GKS84	"Graphical Kernel System," published as a special issue of *Computer Graphics*, February 1984.
GSPC79	"Status Report of the Graphics Standards Committee," *Computer Graphics* 13(3), August 1979.
GUIB82	Guibas, L., and J. Stolfi, "A Language for Bitmap Manipulation," *Transactions on Graphics*, 1(3), July 1982, pp. 191 - 214 .
PRES86	Preston, Asal, Koshell, and Guttag, The Texas Instruments 34010 Graphics System Processor, 6(10).
SALE86	Salesin, D. and R. Barzel, "Two-Bit Graphics", *IEEE Computer Graphics and Applications*, 6(6), June 1986, pp. 36-42.
SHUE86	Shuey, D., D. Bailey, and T. Morrissey, "PHIGS: A Standard, Dynamic, Interaction Graphics Interface," *IEEE Commputer Graphics and Applications*, 6(8), August 1986, pp. 50 - 57.
ZACH84	Zachrisen, Morten, "Adding Structure to Bit-Map Displays," *IEEE Computer Graphics and Applications*, 4(7), July 1984, pp. 47-51.

Appendix A - Composition Expression Syntax

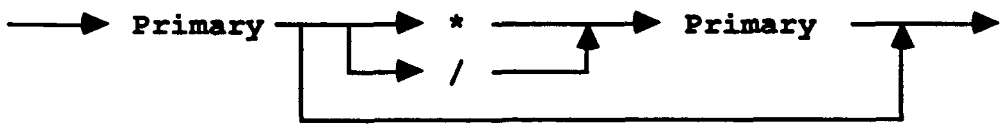
COMPOSITION STATEMENT



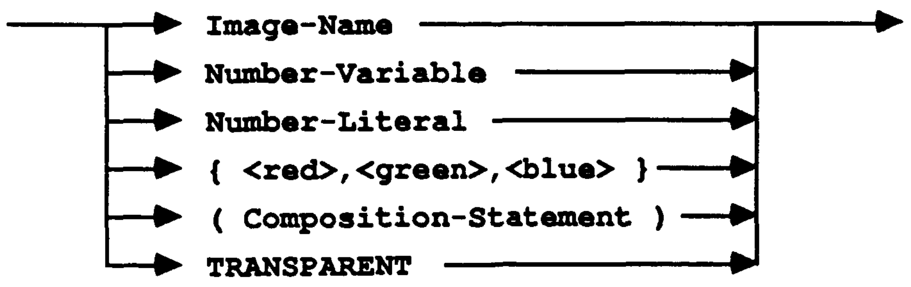
TERM



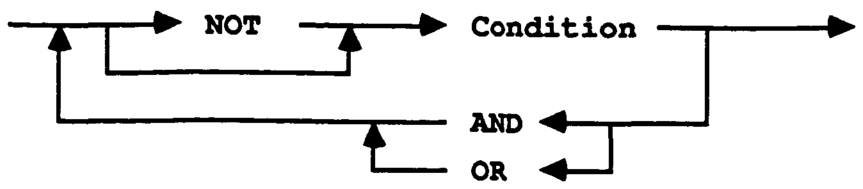
FACTOR



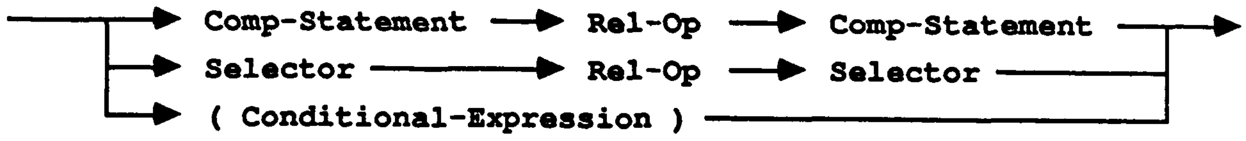
PRIMARY



CONDITIONAL EXPRESSION



CONDITION



RELATIONAL OPERATOR

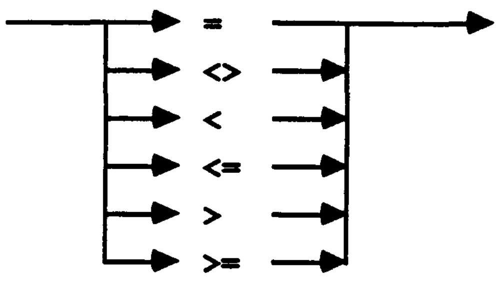


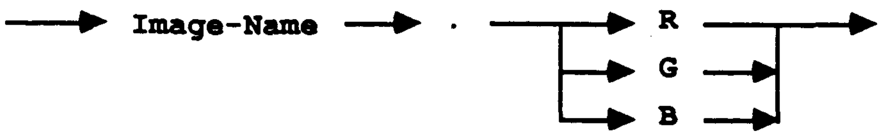
IMAGE NAME , NUMBER VARIABLE

<Identifier>

NUMBER LITERAL, <red>, <green>, <blue>

<Positive Real>

SELECTOR



Appendix B - LUTC subroutine calls.

Type definitions used in these subroutine calls are as follows:

```
type
        PixelRange   = 1..9;
        IndexRange   = 0..511;
        ColorRange   = real;
        Rectangle    = record
                              xmin,xmax,ymin,ymax :  cardinal;
                       end;
```

```
DefineImage             ( ImageName           : array of char  ;
                          BitsPerPixel        : PixelRange     );
```

An image with the given *ImageName* is allocated in the refresh buffer with *BitsPerPixel* planes. *ImageName* may be used in composition expressions. The height and width of the image cover the entire refresh buffer. Error if *ImageName* is already in use of if insufficient planes are free to satisfy the request.

```
DeleteImage             ( ImageName           : array of char  );
```

The named image is deleted, making its refresh buffer planes available. Error if named image does not exist. (This procedure is unimplemented, so that currently, images are statically allocated.)

```
ClearImage              ( ImageName           : array of char  );
```

The image is cleared: all pixels are set to zero. Error in the image does not exist.

```
SelectImage             ( ImageName           : array of char  );
```

The named image is established as the current image to receive output from the graphics subroutine package which sits "on top of" ICL.

```
SetImageIndexColor      ( ImageName           : array of char  ;
                          Index               : IndexRange     ;
                          red,green,blue      : ColorRange     );
```

Entry *index* in the look up table associated with image *ImageName* is set to the given color triplet. Error if the image does not exist, or if *index* is too large (IndexRange is the range of *index* values for the largest possible image).

```
SetImageIndexTransparency ( ImageName         : array of char  ;
                            Index             : IndexRange     ;
                            Trans             : boolean        );
```

Entry *Index* in the look up table associated with image *ImageName* is set to be **transparent**. Error if the image does not exist, or if *index* is too large (IndexRange is the range of *index* values for the largest possible image).

```
DeclareVar                 ( VariableName            : array of char  ;
                             VariableAddress         : address        );
```

A scalar variable used in the Modula program is declared to the look-up table compiler so that it can be used in composition expressions. *VariableName* is the name of the variable as used in composition expressions (it would normally be the same as the Modula-2 variable). *VariableAdress* is the address of the variable, and is returned by the Modula-2 function **adr(***ModulaVariable*).

```
DefineCompositionFrame     ( FrameName               : array of char  ;
                             Bounds                  : Rectangle      ;
                             CompositionExpression   : array of char);
```

This procedure allows the user to declare a composition frame, and the composition expression which is to be used within the composition frame. Error if *FrameName* is already in use, or if the bounds are out of range. Error if *CompositionExpression* violates the syntax rules for composition expressions, references images which have not been defined, or references variables which have not been declared.

```
DeleteCompositionFrame     ( FrameName               : array of char  );
```

The composition frame is deleted. Error if the frame does not exist. (This procedure is unimplemented, so that currently, frames are statically allocated.)

```
ReduceCompositionFrame     ( FrameName               : array of char  ;
                             Bounds                  : Rectangle      );
```

The composition frame region is reduced by the specified rectangular region. Error if the frame does not exist, or if the bounds are out of range.

```
ExtendCompositionFrame     ( FrameName               : array of char  ;
                             Bounds                  : Rectangle      );
```

The composition frame region is extended by the specified rectangular region. Error if the frame does not exist, or if the bounds are out of range.

```
DisplayComposedImage;
```

The composed images defined by all of the currently-existing frames are displayed. If two frames overlap, the more-recently defined one has precedence.

```
SetGlobalBackground        ( red,green,blue          : ColorRange     );
```

The color displayed in areas covered by no frame is set. The default is black, {0, 0, 0}.

```
SetFrameBackground         ( FrameName               : FrameRange     ;
                             red,green,blue          : ColorRange     );
```

The color displayed within the named frame wherever the composition expression evaluates to transparent is set. The default is black, {0, 0, 0}. Error if frame has not been defined.

Captions for Photographic Images

Figure 1. *ImageA*, used to illustrate composition expressions. The vertical stripes have intensities of 1.0, .75, .50, .25, and 0.0.

Figure 2. *ImageB*, used to illustrate composition expressions. The intensities are the same as in Figure 1.

Figure 3. The sum of the two images, *ImageA+ ImageB.*

Figure 4. The difference of the two images, *ImageA- ImageB.*

Figure 5. Images *a* and *b* combined with the **max** operator, *ImageA* **max** *ImageB.*

Figure 6. Images *a* and *b* combined with the **min** operator, *ImageA* **min** *ImageB.*

Figure 7. The image *Mona_Lisa*, a digitized black and white halftone photo.

Figure 8. Expansion of the range [0.5, 1.0] into the range [0.0, 1,0], using the compostion expression (*Mona_Lisa* - 0.5) * 2.

Figure 9. The half-tone image *Mona_Lisa* contoured at an intensity level of .6, so that all values greater than .6 are white, and the rest are black. The composition expression is: **if** *Mona_Lisa* > 0.6 **then** 1.0 **else** 0.0 **endif.**

Figure 10. The *Mona Lisa* image modified using a nested **if ... then ... else** statement.

Figure 11. A graytone image *c*

Figure 12. The result of applying a nested **if ... then ... else** statement with **or** used in the conditional expression.

Figure 13. The result of applying a nested **if ... then ... else** statement with **and** used in the conditional expression to specify ranges of values.

Figure 14. Text strings with filled interior.

Figure 15. Text strings with empty interior, created by subtracting the interior color from the original image.

Figure 16. (not a photo)

Figure 17. The effect of prioritizing three monochrome (one-bit) images on top of one another with a nested **if ... then ... else** statement.

Figure 18. Two overlapping frames with different composition expressions. The frames were created with the **extend** and **reduce** procedures.

Figure 19. A lap-dissolve sequence, showing the effects of the parameter *t* at 0.0, 0.2, 0.4, 0.6, 0.8, and 1.0.