# The Fill Interpreter: A Unified View of Brushing, Filling, and Compositing

*Shawn R. Neely, Kellogg S. Booth, and Peter P. Tanner*

Computer Graphics Laboratory, Department of Computer Science
University of Waterloo, Waterloo, Ontario, Canada N2L 3G1
Tel: 519/888-4534, E-Mail: KSBooth@cgl.waterloo.edu

## Abstract

A unified model for the brushing, filling, and compositing operations found in a variety of computer animation and paint systems is introduced and a specification language based on the model is presented. Extensions to the standard definition of fill are described, including operations where the region boundary and colour of the affected pixels depend on generalized pixel attributes. These are supported by the virtual frame buffer abstraction underlying the model. A linguistic device for specifying comparison tolerances among pixel attributes provides a mechanism for achieving effective results with complex images. An interactive interpreter for the language has been implemented, serving as an exploratory testbed. Examples demonstrating the power of the model and the use of the interpreter are presented.

**Keywords:** brush, composite, fill, paint, virtual frame buffer.

## Introduction

Computer animation and paint systems commonly include facilities for modifying raster images using three operations: brushing, filling, and compositing. We are interested in a family of operations that can be performed on digital images all of which use the abstraction of a *virtual frame buffer* [7] in which the *raster image* or *frame buffer* is a two-dimensional array of *pixels* each having a number of *attributes*. In the simplest cases, only the three *red*, *green* and *blue* colour attributes and a fourth *opacity* or *alpha* attribute are used (these four attributes are collectively abbreviated RGBA). More sophisticated operations use other pixel attributes defined within the virtual frame buffer model.

In *brushing*, a raster image called the *canvas* is modified under the control of a tablet stylus or other pointing device. Often the *brushing algorithm* involves replacing a small region of the image surrounding the current tablet position with another image called the *brush*. More elaborate brushing algorithms define combinations of the brush and the canvas, resulting in new pixel values for the canvas that are derived from the RGBA pixel attributes in the canvas and the brush [22].

Many paint programs also provide a facility for changing the colour of a topologically connected region by *filling* [20]. The colour change follows specific rules and the region is determined by both a *seed point* and a set of rules for propagating the region starting at the seed point. Tools that provide a fill capability are often used to modify certain attributes of the colour in regions of images that have been produced using other graphics techniques, such as images introduced to the frame buffer from photographic or video media, images created by rendering a geometric model, or images created by other tools in a paint program.

*Compositing* is the technique of combining two or more raster images using pointwise rules based on the pixel values in the input images. A complete calculus of compositing operations using the RGBA attributes was introduced by Porter and Duff [18] and extended to additional pixel attributes by Duff [2]. The most common application of compositing is to merge a foreground image with a background image using auxiliary opacity attributes in the two images, making the RGB values of the output image appear as if they were the result of photographically overlaying the two images (as is done in traditional cel animation).

Although brushing, filling, and compositing can be viewed as distinct operations, we believe it is useful to consider them as variants of a more general technique. To support this view, we observe the following "reductions" that demonstrate the conceptual equivalence of the three operations.

*Brushing* $\subseteq$ *Compositing:* Higgins and Booth have considered brushing as a special case of compositing, where the canvas is the background image and the brush is the foreground image [7]. As the position of the tablet stylus changes, the brush image is considered to be translated with respect to the canvas prior to the compositing operation – a weighted sum

of the foreground and background images determined by their opacity attributes.

In the Palette system implemented by Higgins, extended brushing operations involve a canvas with explicit foreground and background images so that the brushing operation becomes a three-level composition of the brush with both the canvas foreground and the canvas background. A separate *mask* attribute can be used at each pixel to further control brushing by inhibiting composition where the mask has high opacity. More elaborate brushing algorithms can be incorporated into this view by permitting the replacement colour to depend on more complicated functions of the RGBA attributes of the brush and canvas.

*Filling* $\subseteq$ *Brushing:* In the literature, filling has been extended from its basic definition to include soft-edge and pattern fills [3] [12] [13] [20], but it has remained conceptually disjoint from brushing operations. We take a broader view here. Filling can be seen as a special case of brushing. Using the notion of masking introduced in Higgins's Palette system, the standard fill operation becomes just brushing with a full-screen brush consisting entirely of pixels having the colour of the fill, with a mask attribute in the canvas having zero opacity inside the fill region and maximum opacity outside the fill region. The mask can have intermediate values for opacity along the boundary of the region if soft edges are desired, and may, in general, have values dependent upon attributes of canvas pixels.

In this model, the mask attribute for each pixel in the canvas must be pre-computed by an algorithm that determines connectivity with the seed pixel, or generated on-the-fly during the brushing operation.

*Compositing* $\subseteq$ *Filling:* To complete the equivalence between the three operations, we observe that compositing is just a fill operation if we consider the canvas to comprise two sets of pixel attributes, one from the foreground image and one from the background image, with the replacement colour for each pixel being computed using the RGBA attributes from both. In this situation, the connectivity computation is trivial because the entire raster is filled.

These remarks are meant to motivate our claim that brushing, filling, and compositing are variants of a single more general set of raster operations. As paint program capabilities have increased, there has been a parallel increase in the functionality of brushing and filling tools, but almost always as independent operations. We believe that a unified view is worth taking because of the flexibility it provides in defining new operations that extend the basic ones.

A unified approach has been taken before in systems that define raster manipulations based on symbolic bitmap expressions. Guibas and Stolfi defined a bitmap calculus in which all of the common brushing, filling, and compositing operations could be defined [6]. Paeth and Booth implemented a raster toolkit capable of performing these operations using Unix pipes between processes that manipulate the raster images according to primitive operations [15]. Nadas and Fournier [14] and Potmesil and Hoffert [19] described more complete environments that encompass similar tools. Common to these earlier systems was a "batch" treatment suggesting composition of static images.

Our work concentrates more on the interactive nature of brushing and filling that arises naturally from a consideration of paint programs. Closer to this approach is work by Perlin [16] and Holzmann [8], where interpretive languages are defined for manipulating raster images by describing pixel-wise operations that are applied to an entire image. The system described here extends this approach to a full virtual frame buffer model and operations that depend on interactive parameters such as tablet $(x,y)$ position so that the operations found in paint systems can be implemented directly.

The sections that follow describe a formal model for a generalized fill algorithm, the implementation of a *fill interpreter* that supports brushing, filling, and compositing in a uniform manner based on the model, and examples of the generalized fill algorithm that have resulted from this work.

## A Formal Model

We assume throughout the discussion the use of a virtual frame buffer model in which each pixel contains an arbitrary collection of attributes or fields, and that a mapping is made from the virtual frame buffer to a physical frame buffer by some projection of the various attributes onto the RGB triplets of the physical device. In addition to opacity (A), depth (Z), and mask (M) attributes, the virtual frame buffer may contain multiple RGB fields as well as many other attributes including surface normal information, quantities resulting from intermediate calculations, and "mark" fields derived from region propagation algorithms. A justification of the virtual frame buffer is given by Higgins and Booth, who also provide details on efficient software implementations of the virtual-to-physical mapping [7].

Fishkin and Barsky have shown the utility of a general filling procedure [3]. Their model includes two components: a Boolean function INSIDE$(x,y)$ that is TRUE if and only if the pixel at $(x,y)$ has some property $P$ and *has not yet been visited by the propagation algorithm* and a procedure SET$(x,y)$ that is performed exactly once for each pixel in the region.

The key insight is that filling consists of two distinct steps: determining which pixels to fill and determining how to fill those pixels.

The propagation algorithm is defined so that the seed pixel for a fill satisfies INSIDE. Neighbouring pixels are then visited *in some unspecified order* to decide if they satisfy INSIDE, with the SET operation being performed on each pixel as it is visited.

In theory, the predicate $P$ can depend on any attribute of the pixel at location $(x,y)$ or on properties of neighbouring pixels (Smith's definition of tint fill appears to rely on this [20]). Unfortunately, this can lead to situations in which the outcome of the INSIDE test will depend critically on the order in which pixels are visited by the propagation algorithm. Fishkin and Barsky discuss a restricted set of predicates for which the INSIDE test is independent of the order in which pixels are visited [3]. We adopt this same restriction.

With this restriction, we can define the region $S$ to be filled as the set of pixels having properties $P$ and $Q$ as follows:

$$S = \{p \mid P(p_0,p) \text{ and } Q(p_0,p)\}$$

where $p_0$ is the seed point and $P$ depends only on attributes of the pixels $p_0$ and $p$ (this is what makes INSIDE independent of the order of propagation). $Q$ is true if and only if there is a "path" between the two pixels consisting entirely of pixels for which INSIDE is true. As such it defines *connectivity* in a topological sense.

The connectivity predicate $Q$ is usually predetermined within the fill algorithm. The predicate 4-CONNECTED, in which each pixel $p$ is connected to four neighbours each sharing an edge with $p$, is typical, although some fill algorithms use an 8-CONNECTED connectivity rule in which pixels are diagonally connected as well. Other rules such as "have the same y-coordinate" are also possible.

The propagation algorithm assumes responsibility for ensuring that each pixel in the region is SET exactly once. This may involve data structures containing "pixels that have been visited" and/or "pixels to be visited" (the current implementation maintains a bit-per-pixel matrix and a stack, respectively). This paper does not address the propagation method. Fishkin and Barsky provide a good analysis of several algorithms [4].

By imposing the restriction that $P(p_0,p)$ depends only upon properties of $p_0$ and $p$, analysis and implementation become much simpler. This rules out the possibility of using properties of neighbours of $p$ or random elements to define the region $S$, but gives instead a context-free model within which to work. The procedure for deciding whether or not $p$ is a candidate for membership in $S$ becomes *deterministic* in the sense that it can be made without regard to how we chose $p$.

We can use this assumption to speed up our computations in two ways. First, we can apply the procedure SET to pixels as they are discovered by the propagation algorithm (assuming that proper bookkeeping has been done). A good propagation method will not stray away from the region $S$, and in this way we can perform the minimal number of SET operations. Many current compositing tools, for example, operate upon an entire image (or rectangular bounding region) even if only a few pixels change. The second advantage is that parallelism is readily possible. We may be able to discover pixels in $S$ in parallel, such as by having four processors explore quadrants about the seed point.

### The Fill Language and Interpreter

The *fill interpreter* is a mechanism for designing and implementing general fill operations. A user specifies the predicate $P$ and a replacement procedure SET. The fill interpreter automatically supplies the connectivity predicate $Q$. (We use 4-CONNECTED exclusively in our examples.)

Both $P$ and SET are specified in the *fill language* which resembles C (in fact we invoke the C preprocessor), with the following exceptions:

1. One data type – all variables, constants, and expressions are *real* No attempt has been made to specify range or precision, although it is assumed that arithmetic on "small" integers is performed exactly.

2. Autodeclaration – there is no need to declare variables before use. Storage is automatically allocated whenever a new variable appears in a program.

3. Control structures, operators, and built-in functions – we implement most of the C control structures, assignment operators, arithmetic operators, logical operators, and functions from the math library. A simple modulus operator for real expressions is included: $a\%b$ is defined as $a-\lfloor a/b \rfloor \times b$.

4. Fuzzy comparison – as Smith notes, thresholding is required in deciding "colourness"; some tolerances about this threshold are necessary [21]. We address this with the introduction of *fuzzy comparisons* in the fill language. A comparison tolerance variable which we call `fuzz` is defined, modifying the semantics of the comparison operators to be as follows:

| expression | equivalent C expression |
|---|---|
| `a == b` | `abs(a-b) <= fuzz` |
| `a != b` | `abs(a-b) > fuzz` |
| `a < b` | `a + fuzz < b` |
| `a >= b` | `a + fuzz >= b` |
| `a > b` | `a > b + fuzz` |
| `a <= b` | `a <= b + fuzz` |

All comparisons use the current value of `fuzz` (which is initially zero). When `fuzz` equals zero the standard semantics (as defined in C) of the comparison operators are preserved. Although we are currently using `fuzz` as an absolute threshold, a mechanism for specifying relative tolerances is under consideration. A similar mechanism exists for APL [9] [11], where typically fuzz is set to a very small value (say $1.0^{-13}$) and used to disguise the fixed-precision representation of real numbers. Our use of fuzz is much more dynamic.

With these definitions, the following familiar properties hold for all values of fuzz:

| expression | equivalent expression |
|---|---|
| `a == b` | `!(a != b)` |
| `a != b` | `!(a == b)` |
| `a < b` | `!(a >= b)` |
| `a < b` | `(a <= b) && (a != b)` |
| `a >= b` | `!(a < b)` |
| `a >= b` | `(a > b) \|\| (a == b)` |
| `a > b` | `!(a <= b)` |
| `a > b` | `(a >= b) && (a != b)` |
| `a <= b` | `!(a > b)` |
| `a <= b` | `(a < b) \|\| (a == b)` |

In practice, these definitions and properties give the user a simple yet powerful model that replaces strict equality with the notion of containment in an interval. We cannot have everything, however; the fundamental transitive property of equality is lost. Fortunately, this has not posed any problems in our applications.

A parser translates a program into a stack-based intermediate language, details of which are beyond the scope of this paper. The fill program comprises two sections – one for *P* and one for SET. A simple interpreter executes the first section whenever *P* is to be evaluated and executes the second section whenever the replacement procedure SET is required.

The current implementation identifies the following fields within a virtual pixel:

| field | associated attribute |
|---|---|
| `X` | column coordinate |
| `Y` | row coordinate |
| `R` | red colour component |
| `G` | green colour component |
| `B` | blue colour component |
| `A` | alpha (opacity) |
| `H` | hue |
| `S` | saturation |
| `V` | value |

A field is referenced like a C structure. For example, `p.R` denotes the red component of pixel *p*. Fields `X` and `Y` have integer values, while all other fields are expected to have values in the range [0,1].

As an example, a tint fill would be completely specified by the following two parts (each a fill language block):

```
{
  fuzz = 0.1;
  inside = (this.H==seed.H &&
            this.S==seed.S);
}

{
  this.H = 0.25;
  this.S = 0.8;
}
```

The first block provides the test that will be applied to every pixel connected to the seed point. The built-in variable `inside` (initially FALSE for each pixel) determines whether or not the pixel passes the test. Built-in variables `seed.<attribute>` are automatically initialized to the values of the seed pixel. Similarly, built-in variables `this.<attribute>` are keywords giving the values of the current pixel (the one being tested), and may be examined and optionally modified. In this example, the `H` and `S` fields refer to hue and saturation attributes, respectively. The built-in variable `fuzz` is a programming convenience (described above) used for the equivalence tests. Setting `fuzz` to 0.1 causes the inclusion test to accept pixels whose tint is close to but not exactly the same as that of the seed point.

The pixel change routine is in the second block. The values 0.25 and 0.8 are the hue and saturation of the replacement tint. In this case, the values in `this.H` and `this.S` (along with the unchanged `this.V`) are used for an HSV to RGB transformation, and `this` pixel is then displayed using the resulting RGB value.

The language provides automatic consistency updates between RGB and HSV representations (change of colour basis). Thus the programmer may use either representation, or a mixture of the two, and still be sure that appropriate transformations are performed whenever necessary. This feature permits simple expression of compound replacement procedures, such as "make the pixel twice as red and then desaturate it by ten percent".

### Some Examples

This section provides a few simple examples of applications of the fill language. The first two examples define typical fills.

*Example 1:*

```
/* basic flood fill INSIDE *{
  inside = (this.R==seed.R &&
            this.G==seed.G &&
            this.B==seed.B);
}

/* SET colour in region to full magenta */
{
  this.R=1.0; this.G=0.0; this.B=1.0;
}
```

*Example 2:*

```
/* INSIDE looks for "same" hue */
{
  fuzz = 0.1;
  inside = (this.H == seed.H)
}

/* SET shifts the hue by 120 degrees */
{
  this.H = (this.H + 0.333) % 1.0;
}
```

The next two examples illustrate the use of the fill language to define brushing algorithms. In the first case $P$ defines an 11x11 square region about the seed point and SET defines a transformation from full colour to NTSC black and white. The fill interpreter continuously samples the current tablet $(x,y)$ to determine a seed point, which is then filled (brushed).

*Example 3:*

```
/* INSIDE is a square brush */
{
  fuzz = 5;
  inside = (this.X==seed.X && this.Y==seed.Y)
}

/* switch to NTSC black and white */
{
  y = this.R*0.30 + this.G*0.59 + this.B*0.11;
  this.R = this.G = this.B = y;
}
```

In the second case a circular brush of radius 10 pixels is defined inside which pixels are converted to high contrast black and white images by setting the saturation to zero (this makes setting the hue irrelevant) and the value to either full intensity or no intensity, depending on a random threshold.

*Example 4:*

```
/* INSIDE is round brush */
{
  radius = 10;
  dx = this.X - seed.X;
  dy = this.Y - seed.Y;
  inside = (dx*dx + dy*dy) < (radius*radius);
}

/* SET is random dither to single-bit B&W */
{
  this.S = 0;
  this.V = (random() > this.V) ? 0 : 1;
}
```

The built-in function `random()` provides a uniform distribution on the interval [0,1]. A library of mathematical functions is also provided.

Any of the examples could have used the following SET definition instead to achieve a hue that depends on the $(x,y)$ position of the pixel.

*Example 5:*

```
/* SET for rainbow stripe */
{
  this.H = ((this.X + this.Y) % 256) / 255;
  this.S = 1;
  this.V = 1;
}
```

The next example defines a "compositing brush" which blends two images along the path of the tablet stylus. The brush is rectangular, and illustrates the use of different fuzz values. (Alternatively, we could have simply set `inside` to TRUE, affecting the entire frame.)

*Example 6:*

```
/* INSIDE for 11x7 compositing brush */
{
  inside = FALSE;
  fuzz = 5;
  if( this.X == seed.X )
    {
      fuzz = 3;
      if( this.Y == seed.Y )
          inside = TRUE;
  }
}

#define blend(x, y, a)  (a)*(x)+(1-(a))*(y)

/* SET assumes "fore" and "back" #defined */
{
  this.R = blend( fore.R, back.R, fore.A );
  this.G = blend( fore.G, back.G, fore.A );
  this.B = blend( fore.B, back.B, fore.A );
}
```

The final example illustrates the use of the fill interpreter to build an interactive tool for examining the contents of the frame buffer. The *P* predicate doesn't select any pixels, but instead simply prints out the $(x,y)$ position of the tablet and the RGBA contents of the virtual frame buffer at the pixel. Because *P* is never true (`inside` is initially FALSE), it is only executed for the seed point. The SET function is null in this case.

*Example 7:*

```
/* attribute inquiry */
{
  print( "x = ", this.X, "y = ", this.Y );
  print( this.R, this.G, this.B, this.A );
}

/* SET is null */
{
}
```

More elaborate examples of the fill interpreter are illustrated in the figures at the end of the paper.

Figure 1 (courtesy of Michael Sweeney) shows a face rendered using a ray-tracing algorithm. Figure 2 is the result of changing the hue at each pixel within the face to a straw colour and then modulating the value at each pixel using a weave pattern, similar to the rainbow stripe example above. Figure 3 is the result of further modulating the alpha (opacity) of the weave pattern and compositing the resulting image over two coloured rectangles.

Figure 4 shows a line drawing of various kitchen utensils produced by a CSG modeler. Figure 5 is the result of a basic fill applied to the cutting board, rolling pin, and spoon, with gradient fills (value is a function of $y$) applied to the bowl, eggs, and table. Artificial highlights were then added to the bowl using a single application of a brush that intensifies the seed point and rapidly diminishes as it moves away from the seed point. A brick pattern fill was performed on the wall, which was then modified by individual applications of a random colour shift to various bricks. A few bricks were filled with random grey and graffiti was brushed on using a circular brush having a Gaussian interpolation parameter.

Figure 6 is an example of abstract artwork created using only the filling tool. An $n$-pointed star was created by converting to polar coordinates and constraining the fill to $r \leq sin(\theta)$.

Examples of *z-fill* in which an explicit depth attribute is used to control filling/brushing, *specular brushes* in which saturation is a function of a surface normal attribute, *composite brushing* in which two images are merged in a region defined by tablet strokes, and many more are easily specified with the fill language.

## Implementation

The fill interpreter is portable and has been implemented for two different hardware configurations so far.

The first is a multiprocessor configuration in which a VAX 8600 compiles the fill program for downloading into an Adage bitslice processor. The interpreter resides in the bitslice, where arithmetic operations are performed in a scaled fixed-point format. Cursor tracking, seed point selection, and other user interface duties are provided by a 68000 microprocessor running multiple tasks under the Harmony operating system [5]. In this configuration, one can alternately run the fill interpreter and the multitask Harmony-based Paint system [1].

The second configuration has both compiler and interpreter residing on the 8600, with the user interface provided on a VAXstation II/GPX running the X Window System. Single-precision floating-point format is used.

Both configurations use frame buffer memory to store pixel arrays. R,G,B, and A fields are stored explicitly, while H,S,V fields are computed as required [20]. Performance is better on the multiprocessor configuration, but both systems are

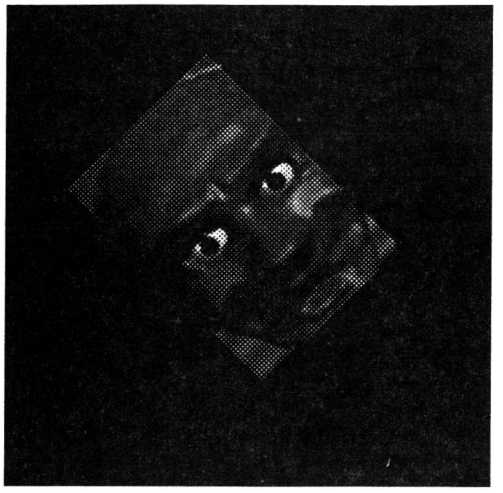

Figure 1

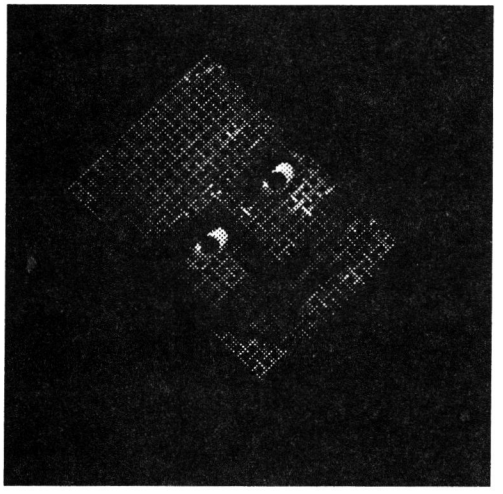

Figure 2

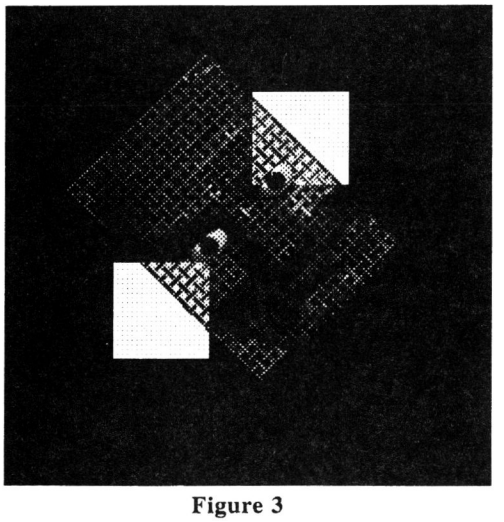

Figure 3

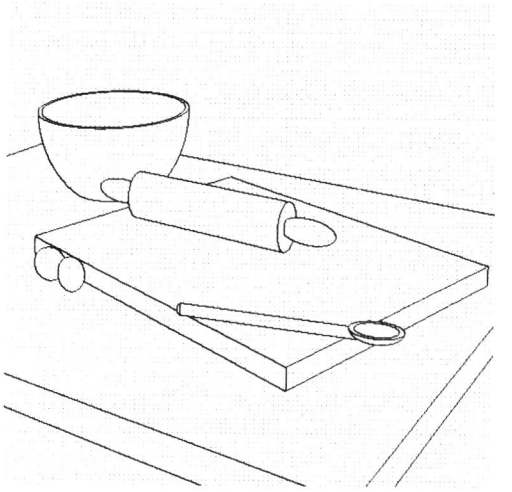

Figure 4

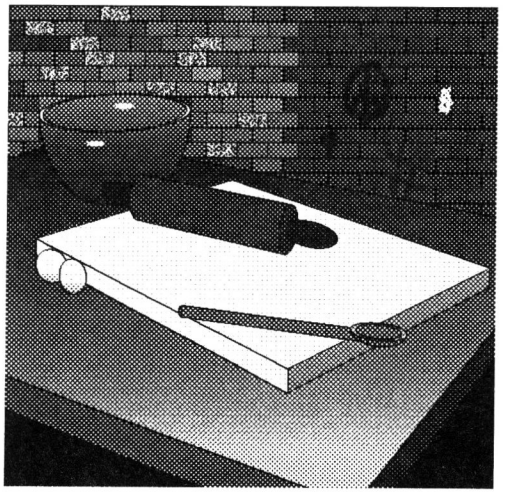

Figure 5

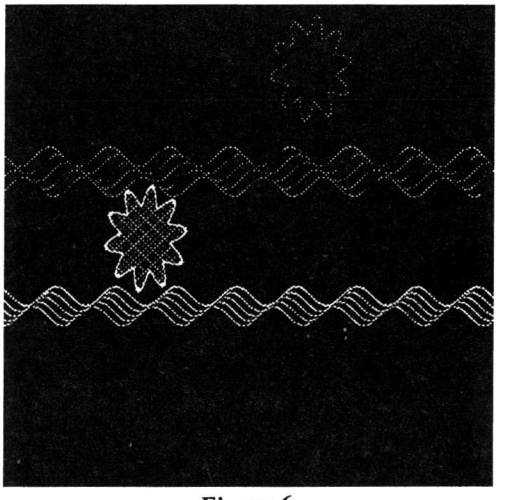

Figure 6

capable of real-time activity. For production work, fill programs are automatically translated into C subroutines for compilation into dedicated programs.

A number of optimizations remain to be implemented in the fill interpreter, such as run-time compilation of efficient field handling routines expressed directly in machine code [10] [17], rather than the interpreted intermediate language used currently. The automatic conversion between RGB and HSV representations relieves the user of tedious bookkeeping, but the current implementation performs the conversion more often than necessary. Standard compiler techniques for flowgraph analysis could be used to minimize these conversions.

### Discussion

It has been observed that brushes commonly in use in today's paint systems, including all brushes based on the rubber-stamp approach, can be implemented by the fill interpreter. As a programmer's tool it is excellent for discovery, but it is not yet appropriate for artists whose expertise lies outside the realm of computing.

To make the results of this work accessible to artists, paint tools developed with the fill interpreter need to be classified and parameterized. A possible interface approach would be "cafeteria-style", where an artist could assemble a painting tool by choosing attributes implemented (and perhaps discovered) through the use of the fill interpreter. INSIDE factors such as shape, size, or colour conditions and SET functions affecting tint, value, or texture of the region must be selectable. Significant work remains to be done to organize the tool parameters into a comprehensive and comprehensible hierarchy.

Experience with the fill interpreter has shown it to be a useful tool for experimenting with different ways to modify the shade or texture of areas within a frame-buffer image. Elaborate effects may be programmed quite simply. The ease with which this is accomplished using a single tool blurs the distinction between brushing, filling, and compositing.

### Acknowledgements

This work was supported by an operating grant from the Natural Sciences and Engineering Research Council of Canada and by equipment and operating funds from Digital Equipment of Canada.

### References

[1] R. J. Beach, J. C. Beatty, K. S. Booth, E. L. Fiume, and D. A. Plebon. The message is the medium: Multiprocess structuring of an interactive paint program. Proc. SIGGRAPH '82 (Boston, July 26-30 1982.). *Computer Graphics*, 16(3):277-287, July 1982.

[2] T. Duff. Compositing 3-D Rendered Images. Proc. SIGGRAPH '85 (San Francisco, July 22-26 1985.). *Computer Graphics*, 19(3):41-44, July 1985.

[3] K. P. Fishkin and B. A. Barsky. A Family of New Algorithms for Soft Filling. Proc. SIGGRAPH '84 (Minneapolis, July 23-27 1984.). *Computer Graphics*, 18(3):235-244, July 1984.

[4] K. P. Fishkin and B. A. Barsky. An Analysis and Algorithm for Filling Propagation. In *Proc. Graphics Interface '85*, pages 203-212, May 1985.

[5] W. M. Gentleman. *Using the Harmony Operating System.* Technical Report NRCC-ERB-966, Division of Electrical Engineering, National Research Council of Canada, December 1983.

[6] L. J. Guibas and J. Stolfi. A Language for Bitmap Manipulation. *ACM Transactions on Graphics*, 1(3):191-214, July 1982.

[7] T. M. Higgins and K. S. Booth. A Cel-based Model for Paint Systems. In *Proc. Graphics Interface '86*, pages 82-90, May 1986.

[8] G. J. Holzmann. PICO - A Picture Editor. *ATT Technical Journal*, 66(2):2-13, March/April 1987.

[9] H. Katzan, Jr. APL User's Guide. Van Nostrand Reinhold, 1971.

[10] K. C. Knowlton. A Programmer's Description of $L^6$. *Communications of the ACM*, 9(8):616-625, August 1966.

[11] R. H. Lathwell. APL Comparison Tolerance. Proc. APL '76 (Ottawa, September 22-24 1976.)

[12] M. Levoy. *Area Flooding Algorithms.* Hanna-Barbara Productions, June 1981. Reprinted in SIGGRAPH '82 2-D Animation Tutorial notes.

[13] H. Lieberman. How To Color In A Coloring Book. Proc. SIGGRAPH '78 (Atlanta, August 23-25 1978.). *Computer Graphics*, 12(3):111-116, August 1978.

[14] T. Nadas and A. Fournier. GRAPE: An Environment to Build Display Processes. Proc. SIGGRAPH '87 (Anaheim, July 27-31 1987.). *Computer Graphics*, 21(4):75-84, July 1987.

[15] A. W. Paeth and K. S. Booth. Design and Experience with a Generalized Raster Toolkit. In *Proc. Graphics Interface '86*, pages 91-97, May 1986.

[16] K. Perlin. An Image Synthesizer. Proc. SIGGRAPH '85 (San Francisco, July 22-26 1985.). *Computer Graphics*, 19(3):287-296, July 1985.

[17] R. Pike. Graphics In Overlapping Bitmap Layers. *ACM Transactions on Graphics*, 2(2):135-160, April 1983.

[18] T. Porter and T. Duff. Compositing Digital Images. Proc. SIGGRAPH '84 (Minneapolis, July 23-27 1984.). *Computer Graphics*, 18(3):253-259, July 1984.

[19] M. Potmesil and E. M. Hoffert. FRAMES: Software Tools for Modeling, Rendering, and Animation of 3D Scenes. Proc. SIGGRAPH '87 (Anaheim, July 27-31 1987.). *Computer Graphics*, 21(4):85-93, July 1987.

[20] A. R. Smith. Tint Fill. Proc. SIGGRAPH '79 (Chicago, August 8-10 1979.). *Computer Graphics*, 13(2):276-283, August 1979.

[21] A. R. Smith. *Fill Tutorial Notes*. Technical Memo No. 40, Lucasfilm Ltd. Reprinted in SIGGRAPH '82 2-D Animation Tutorial notes.

[22] A. R. Smith. Paint. In *Tutorial: Computer Graphics*, J. C. Beatty and K. S. Booth eds. (IEEE Computer Society 1982), pages 501-515.