An Integrated Color Smalltalk-80 System

Rebecca Wirfs-Brock
rebeccaw%spt.tek.com@relay.cs.net
(503) 627-5882

P.O. Box 500, Mail Sta. 50-470
Tektronix, Inc.
Beaverton, OR 97077

ABSTRACT

The Smalltalk-80™ user interface and graphics model are based on monochromatic graphics. One natural step in the evolution of the Smalltalk-80 system is the addition of color. This paper describes an implementation of color Smalltalk. Classes have been defined to manipulate visual color models and colored graphics objects. The extensive collaboration between classes which describe color, classes which perform basic graphics operations, and classes in the user interface is explored. Issues in the design and implementation are examined. Potential future directions for object-oriented color systems are discussed.

Introduction

The Smalltalk-80 user interface and graphics model described in [Goldberg and Robson, 1983] are based on a monochromatic graphics imaging operation called BitBlt (bit block transfer). Most commercially available workstations and personal computers use color display hardware. Readily available windowing and graphics systems on these platforms support color. A natural extension of Smalltalk-80 includes the development of color graphics and a color BitBlt imaging scheme.

This paper describes the color Smalltalk system developed at Tektronix. This is our third implementation of color for Smalltalk. The first two implementations were research prototypes where we explored, among other things, the feasibility of specialized hardware support for color [Wirfs-Brock and Miller, 1986]. We approached our third implementation with the strong conviction that color needed to be highly integrated into the existing Smalltalk-80 user interface and graphics paradigm. We wanted to provide a rich framework for the construction of color Smalltalk applications. The Smalltalk programmer should have easily accessible mechanisms for selecting the colors to use in drawing, for experimenting with and generating colorful graphics, and for constructing visually interesting applications.

First we will discuss our objectives for this work and describe the color, graphics and supporting classes in our system. We will explain our color BitBlt imaging model. Next we will show an example of shared responsibilities of and the communications between color, graphics, and utility classes. We will then discuss the framework we built to allow cooperation between color applications. Finally, we will summarize our work and briefly explore possible future directions.

Design Goals

Our design focused on providing solutions to four fundamental questions.

- What are the appropriate graphics objects and imaging model to present to the user?
- How should colors be described and managed?
- What is an appropriate relationship between colors and colored graphics objects?
- What policies should we establish for using color graphics in an application, and what simple mechanisms can we put in place to support these policies?

Our primary objective was to produce a complete color implementation that would be useful to a broad spectrum of Smalltalk-80 programmers. Adding color to an application should be a natural process. The typical Smalltalk programmer does not want to deal directly with the physical representation of bitmaps or limitations of a hardware color map. The programmer with some level of graphics sophistication should have a rich enough set of capabilities for constructing reasonably sophisticated applications.

Our implementation was to be an evolutionary extension of the Smalltalk-80 graphics and user interface environment. We felt that it would simply create too large a gap between monochrome and color if we abandoned the Smalltalk-80 BitBlt, or the Model-View-Controller paradigms in favor of something new.

We also wanted our implementation to be upwardly compatible with monochrome Smalltalk. The transition path for the existing Smalltalk-80 applications to color should be smooth. An application that did not redefine basic graphics and user interface classes by modifying or relying on the structure of instance variables in these classes should run correctly without modification. We wanted to be able to transfer graphics objects between monochrome and color Smalltalk. We therefore developed a formal specification for color BitBlt graphics that was completely upwardly compatible with monochrome graphics. Although we rearranged the display object class hierarchy, we preserved protocol found in monochrome Smalltalk-80, even when a class moved to a new location in the graphics object hierarchy.

Special modes or global state should not intrude into typical programming situations. Our BitBlt imaging model also operates across all types of Forms. This allows one to program free of the need to learn a litany of rules and exceptions for generating graphics.

The Color Framework

A color framework requires a mechanism for describing color, and another mechanism to associate colors with Forms. In order to describe colors, we devised a color model abstraction, and a corresponding set of classes which create and manipulate colors defined within a particular color model. A second mechanism supports grouping colors together to form a palette to use when displaying colored Forms. Colors appearing on the screen are defined by the current values of the hardware palette.

Color Models

A color model is a system that allows for the orderly description or classification of color according to a basic definition. We defined an abstract color class, and concrete classes to support specific color models.

Color is an abstract class which supports creating colors. Figure 1 illustrates the hierarchy.

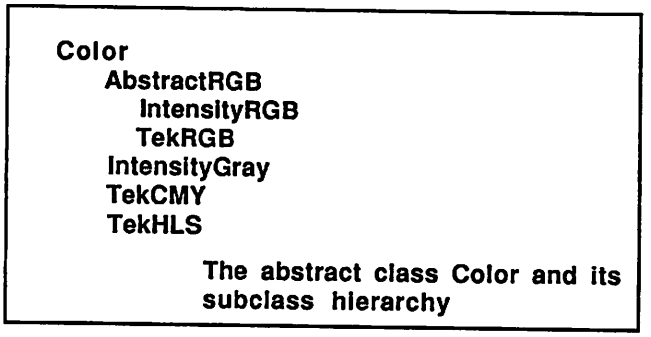


Figure 1

The color models we support are RGB, CMY, HLS, and grayscale.

RGB is an additive color model. In an RGB color model, coloring is thought of as adding colored light. A color is composed of three components representing the contribution of red, green, and blue to that color. Raising the levels of red, green and blue produces a lighter color. Our two RGB classes define the contributions of red, green and blue in terms of linear intensities, or as perceptually even lightness increments. An object-oriented system allows us to clearly define the properties of any RGB color object and refine this abstraction with different concrete representations. For example, we defined both IntensityRGB and TekRGB classes which model different RGB systems.

In the CMY color model, a color is composed of three components representing the contributions of cyan, magenta, and yellow to the color. When colors are mixed in this system, less light is reflected and colors appear darker. CMY is referred to as a subtractive color model and is familiar to artists and those familiar with the printing process.

The HLS color model represents a color as a mixture of the qualities of hue, lightness (the amount of white or black in a color), and saturation (the extent to which a color differs from a gray of the same lightness).

The grayscale color model represents colors as shades of gray of varying intensities between white and black.

A color expressed in any model can be converted to a color expressed in any other model. This allows programmers the flexibility of dealing with colors in terms familiar to them. However, many users are not familiar with colors in terms of their components. Therefore, we added simple protocol to the Color class for creating instances of its concrete classes by referring to common English names. Color and its subclasses understand how to create the base colors black and white and the colors pink, red, orange, brown, yellow, green, blue, purple, magenta, cyan and gray. Lighter or darker versions of a color may be created by prepending 'light' or 'dark' to the capitalized base name, for example, lightBlue or darkGreen.

Palettes and Colors

Mapping between pixels and colors is represented by the Palette class. Pixel values are used as indexes into a palette, which is an array holding information about the color to display for that pixel. A Palette is a collection of colors. A Palette can hold any concrete instance of a color class. The interface to the display color map is represented by a single palette, the HardwarePalette. HardwarePalette is a subclass of Palette. The system hardware palette is controlled by the DisplayScreen object. See Figure 2.

The Display Object Hierarchy

In Smalltalk-80, the class DisplayObject defines an abstraction for classes of objects that can be displayed (or drawn) upon a DisplayMedium. The class DisplayMedium, which is a subclass of DisplayObject, represents an object that can be displayed upon. In Smalltalk-80, DisplayObjects are partitioned (not cleanly in all cases) into classes of objects that can be displayed on (all the subclasses of DisplayMedium) and classes which represent graphical rendering paradigms, such as Circles and Arcs. DisplayMedium subclasses lead a dual life in Smalltalk-80; they are classes which represent fundamental graphics constructs that can also be displayed upon.

A side-by-side comparison of the color DisplayObject and the monochrome class hierarchies illustrates the changes we made.

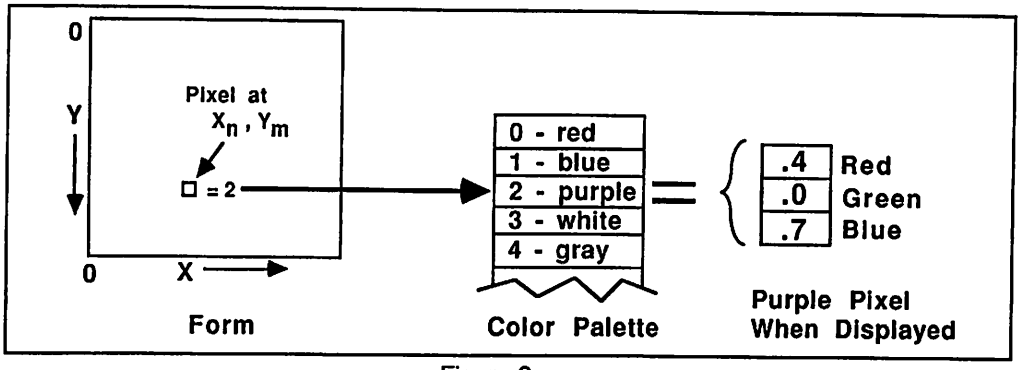
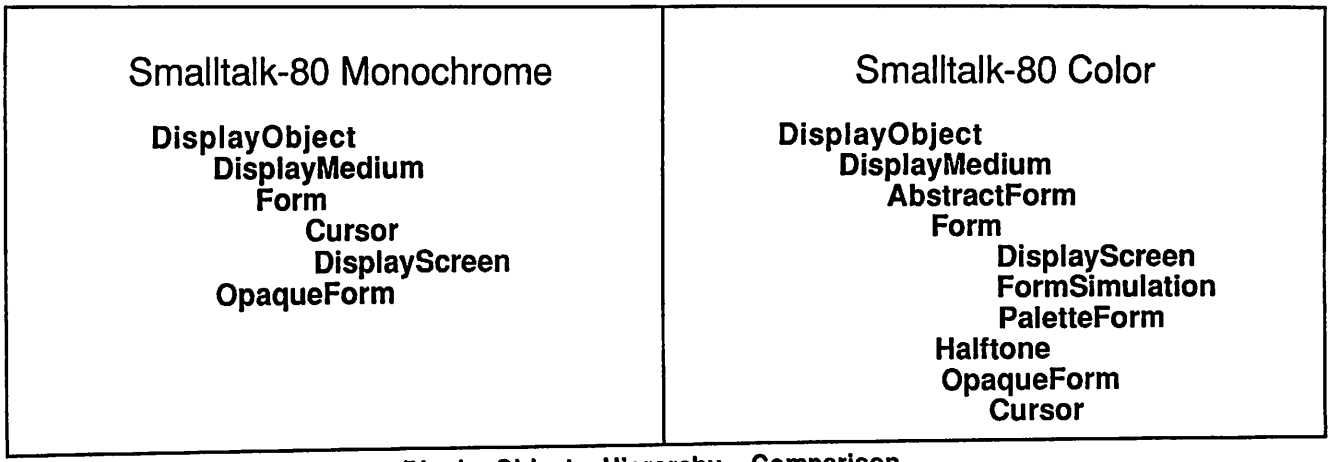


Figure 2



DisplayObject Hierarchy Comparison

Figure 3

Several differences are immediately apparent.

- We moved OpaqueForm to be within the DisplayMedium hierarchy. This added to color Smalltalk the ability to construct OpaqueForms directly by drawing on them, which is possible only if a DisplayObject is also a DisplayMedium. There seemed no compelling reason not to broaden the utility of OpaqueForms, particularly since their very name suggests that they are a variant of Form.
- We also defined a new class, AbstractForm, which defines the abstraction for two-dimensional patterns of pixels. In monochrome Smalltalk, the class Form served as both the abstract model for two-dimensional pixels arrays, and as a concrete representation. The class AbstractForm now serves as the abstract model, and leaves particular representations to its subclasses.
- We also defined two new classes, PaletteForm, and Halftone.

Display Objects

The class AbstractForm provides abstract protocol for subclasses that store images as two-dimensional arrays of pixels. Instances of AbstractForm subclasses can be created by specifying the extent (width and height of the pixel array), and the depth (the number of bits representing a pixel). Instance protocol is defined to edit the contents of the object's pixels, perform basic image manipulation operations such as rotation and scaling, access pixels within the object, ask the chromaticity of the object, and convert a subclass of AbstractForm to any other subclass of AbstractForm. Concrete subclasses of AbstractForm specify explicit representation details. In addition to defining new class and instance protocol, all protocol defined in monochrome Smalltalk is supported for Forms of any depth.

A Form is a two-dimensional rectangular array of pixels. Forms in color Smalltalk have the additional instance variable, depth, which is not present in monochrome Smalltalk. The possible range of pixel values is determined by the depth of a Form. Forms of depth one are equivalent to monochrome Smalltalk-80 Forms. A Form of depth eight, for example, could represent one of 256 different values in each pixel, as eight bits per pixel translates to 2^8 possible values. Forms are typically of depth one or the depth of the DisplayScreen, for practical performance considerations, but are not restricted to those depths. Additionally, color Smalltalk Form instance protocol supports a variety of coloring, dithering, and depth coercion facilities.

The physical display or frame buffer is represented by a single instance of the class DisplayScreen. Because DisplayScreen is a subclass of Form, it is assumed to be defined as a rectangular array of pixels.

A FormSimulation simulates pixel accessing for instances of Form whose depth is not supported by primitive (non-Smalltalk) code. FormSimulation is part of the mechanism which enables us to support graphics operations on forms of any depth.

We could have defined Form to include a Palette, but chose not to. Frequently, a programmer wants simply to deal with a pattern of pixels. At other times, the relationship between pixels and their color needs to be explicitly controlled. Therefore, we defined a PaletteForm to be a subclass of Form with an associated Palette.

A Halftone is the primary mechanism whereby desired color effects can be specified to BitBlt and its subclasses. Halftones are used to achieve a variety of graphics effects, such as stenciling color onto Forms of depth greater than one, or tiling regions of pixels with repeating patterns.

In monochrome Smalltalk there is no Halftone class. Instead halftone masks of extent 16 by 16 are arguments to BitBlt. Monochrome Smalltalk Form class protocol defined constant Forms suitable as BitBlt halftone arguments. In our color implementation, we added much additional color information to the definition of the halftone mask argument to BitBlt. These augmented halftoning capabilities are supported by the Halftone class.

An OpaqueForm is a composite object which stores an image as a Form along with a clipping mask (a Form of depth one). The clipping mask defines whether pixels in the Form are opaque (the corresponding clipping mask bit is one) or transparent (the corresponding clipping mask bit is zero). OpaqueForms allow the display of nonrectangular patterns of pixels. An example of an OpaqueForm might be a cursor or a mailbox icon.

BitBlt and Graphics Imaging

The class BitBlt performs the primary graphics imaging operation in Smalltalk-80, a pixel-by-pixel transfer from a source to a destination form. Instance variables in Smalltalk-80 BitBlt include a source form, a halftone form, a destination form, and a rule

which specifies how pixels are to be combined. In color Smalltalk, the source, halftone or destination form can be any subclass of AbstractForm of any depth. The precise transfer function for pixels is controlled by a number of other BitBlt instance variables.

We had explicit performance goals for certain BitBlt cases. Performance for typical BitBlt operations in the color development environment were to be as fast or faster than our monochrome implementation. Performance of common onscreen graphics operations needed to be optimal. In particular, displaying text, erasing and filling interiors of views, removing and redrawing scroll bars, highlighting text, and framing windows required special attention. Support for combinations of Forms having a depth of one with Forms of other depths was necessary, for example, to display text on the display screen.

We added a source clipping mask instance variable to color BitBlt. This allows us to enable transfer of the source form on a pixel-by-pixel basis when displaying OpaqueForms. Drawing an OpaqueForm is thus supported in a single BitBlt operation, using arbitrary combination rules. Primitive color BitBlt code directly supports drawing OpaqueForms. In contrast, monochrome Smalltalk displays OpaqueForms in a two-step operation: the first BitBlt clears the destination area described by the OpaqueForm shape (or clipping mask), the next BitBlt displays the OpaqueForm figure. Such a two-step process would simply not work for transferring colored pixels. The notion of transparent pixels (source pixels not transferred to the destination form) needs primitive support in a color system. Clearing and then ORing of colored pixels does not produce the desired visual effect as those operations do with monochrome BitBlt. Clearing the destination form changes the values of the pixels within it. Because these values are no longer direct color specifications, but instead are indexes into a palette, changing these values for a color destination simply changes the colors of its pixels. In a color system, the *clearing* operation is simply not defined. Furthermore, using the OR combination rule combines two pixel values to produce a third, often unexpected value.

BitBlt Imaging Effects

A Halftone is a composite object consisting of a Form and a PixelStyle. A pixel style allows the specification of a number of parameters, including a foreground value, a background value, and a filter. These parameters allow a variety of interesting imaging effects, among them filling, stenciling, filtering, and stippling.

A pixel of depth one can only have one of two possible values: one or zero. When pixels of different depths are combined, there must be a mechanism for producing the correct pixel value. *Stenciling* is an operation that transforms pixels of depth one into pixels of depths greater than one. Values which are written to pixels of depth greater than one are controlled by the pixel style. A pixel style contains specifications for foreground and background pixels. The rule is quite simple: pixels of depth one are translated to their corresponding foreground or background pixel value before being combined with pixels of greater depth. The pixel style foreground value corresponds to one in a monochrome pixel, the background value corresponds to zero. For example, characters in a font are stored as monochrome Forms (as is much of the existing user interface) and transferred to the display screen through a BitBlt stenciling operation. See Figure 4. If the background value is nil, background pixels are not transferred to the destination.

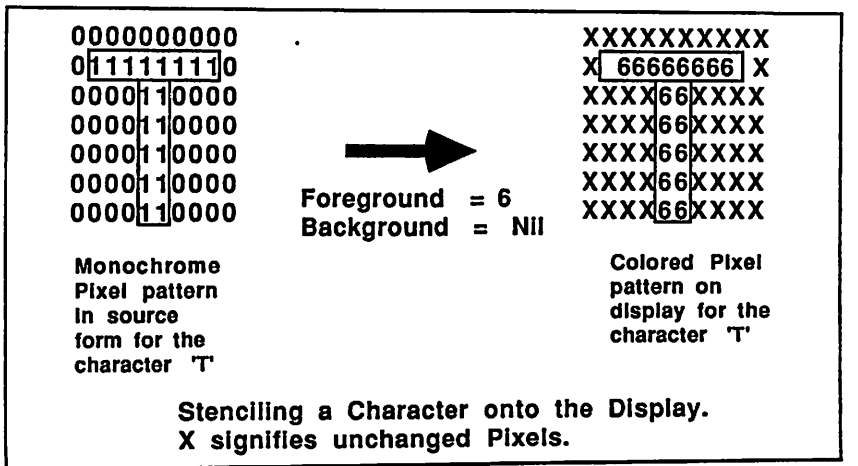


Figure 4

Filtering is an operation that transforms colored pixels into monochrome pixels. See Figure 5. This is also controlled by the pixel style, which contains a filter pixel value. If a colored pixel has the same value as the filter pixel, it will be transformed into a one, if it does not, it will be transformed into a zero. We have used the filtering operation for a variety of applications. For example, filtering can be used to highlight text written in one color on a background of another color.

Filling is an operation whereby colored halftone pixels are directly combined with destination form pixels. The interior of a view is filled by tiling a halftone of depth one (with the appropriate pixel value) onto the view. A pattern can also be repetitively applied to fill the entire destination form by tiling. See Figure 6.

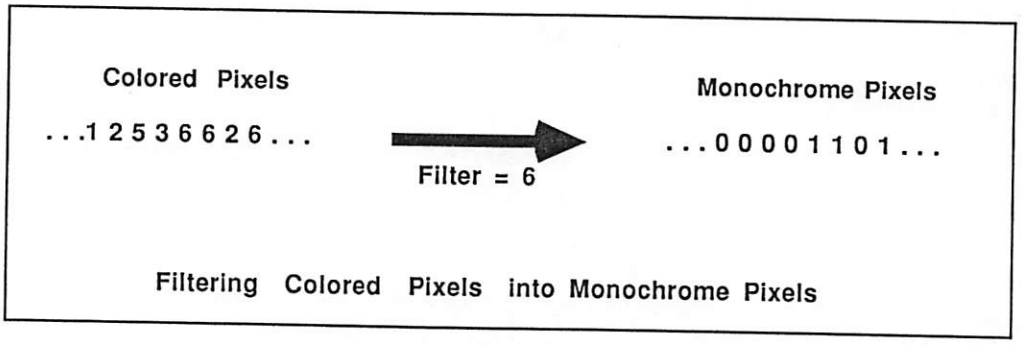


Figure 5

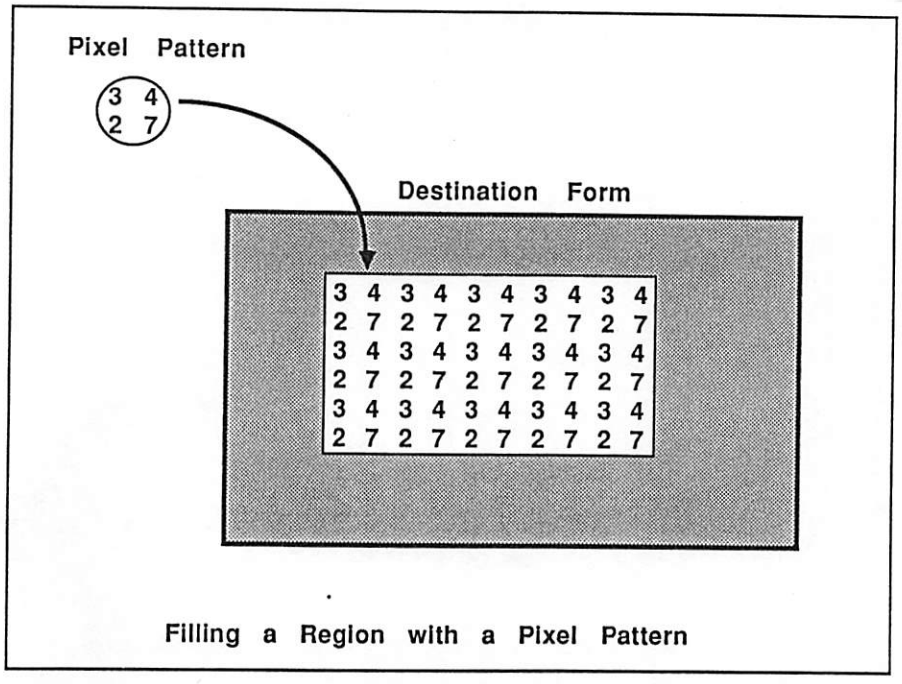


Figure 6

Stippling is an operation that combines colored source form pixels directly with destination form pixels, wherever the halftone form logically ANDed with the source clipping mask is one. Stippling controls translucency and is an effective technique for de-emphasizing an area (for example, de-emphasizing a menu selection when it is currently unavailable to the user).

This very brief overview of BitBlt capabilities is the level of detail most Smalltalk programmers require. We were able to hide further details of BitBlt by providing high-level protocol for displaying objects and generating colors using the previously described techniques. The utility graphics classes play an important role in facilitating such a higher level interface. A more complete description of color BitBlt is given in the appendix.

Dithering

We implemented color Smalltalk on a workstation with a limited hardware palette. To relax this limitation, we developed ColorDither and its subclasses that support color and grayscale dithering. Through dithering, a color is represented as a pattern of colored pixels rather than a single pixel. By trading off spatial resolution for a greater number of displayable colors, more colors can be effectively displayed than can be represented in the hardware palette [Foley and Van Dam1982]. This is the color analog of using varying black and white halftone patterns to obtain shades of gray in a monochrome system. The dither classes turn a Color into a Halftone pattern that approximates that color by spatial averaging among other colors. A Form can also be dithered into a Form of the same width and height but with fewer bits per pixel.

Ordinarily, programmers using color must choose specifically whether to use dithered or nondithered colors. Programmers using our system need not know how colors are generated, nor are they forced to make this choice. Our default hardware palette for color Smalltalk is a palette appropriate for dithering, containing an ordered sequence of the primary colors. These primary colors occupy the upper half of the palette, leaving the lower half of the palette free for other pure, undithered colors. Dither classes implement protocol to return dithered Halftones and keep preinitialized default dither information. Thus programmers can use dithered or nondithered color in high-level graphics operations with no extra programming required.

Object Collaboration in the Color Smalltalk Interface

High level graphics are implemented in our system by several objects with shared responsibilities. Utility classes, such as Halftone or ColorDither, serve an important role. These classes provide a straightforward interface for classes that call them to implement part of the work. The internal details of utility methods, however, assume a great deal of knowledge about other classes in the system. A well partitioned object-oriented design should present a clean interface to classes used directly by programmers. Detailed knowledge about the overall workings of the system (and corresponding message protocol) properly rests with utility classes or within methods intended to be private. As an example of cooperating classes, we will next trace the operation of filling a form with color.

Filling a region with color

The programmer can fill an area with dithered color with the single message fill:color: to a Form object. Let us examine how this works.

```
Display fill: aRectangle color: Color lightPurple
```

Display fills an area using the OVER BitBlt combination rule and a mask or halftone constructed from the specified color, as shown below.

```
self
  fill: aRectangle
  rule: Form over
  mask: (Halftone color: Color lightPurple)
```

The message lightPurple to Color returns a color object with the appropriate components for light purple.

```
Color lightPurple
  "Answer an instance of a concrete Color sub-
  class that represents the color lightPurple."

  ↑self fromIntensityRGB: (IntensityRGB
    red: 0.7 green: 0.2 blue: 1)
```

Sending the message color: to Halftone class constructs a dithered halftone pattern. This dithered halftone is created by consulting dither and color classes. Code in the method for Halftone color: relays to ColorDither default, which returns a color dither object. That color dither object is responsible for generating the halftone pattern.

```
Halftone color: aColor
  "Answer an instance of the receiver with form
  approximating <aColor> and the default pixel
  style. Relay to the default ColorDither."

  ↑ColorDither default color: aColor
```

The color dither object receiving the message color: calls upon Halftone to create an empty halftone.

```
newHalftone ← Halftone form: (Form
  extent: self tilingExtent
  depth: self depth).
```

It next asks Color for an array of color components.

```
components ← aColor asIntensityRGB
  components.
```

Then the color dither proceeds to build the dithered halftone through a series of BitBlt filter operations. The color: method is implemented in some twenty lines of Smalltalk code. After the halftone is

generated, the message fill:rule:mask: can be sent. Code in fill:rule:mask: generates a BitBlt object and sends copyBits, performing the fill operation with the constructed color halftone.

The work of halftone creation was distributed among Halftone, ColorDither, Color and BitBlt objects. The most complicated algorithm, that of actually controlling the generation of the colored halftone pixels was, most properly, performed by a dither object.

A Framework For Cooperative Use of Color

The hardware palette is a limited resource. We devised a technique for sharing palettes among several windows or applications. Let us examine how colors can be shared between applications and the supporting framework provided for such cooperation.

Colors and Views

All views (or windows) in color Smalltalk have a viewStyle instance variable. A ViewStyle consists of a Palette and a PixelStyle. Through these variables, a view has its own colors to display text, background, and other graphics. Whenever a view is activated, its ViewStyle is installed. This loads its palette into the hardware palette, and sets the default PixelStyle to its PixelStyle. The PixelStyle specifies the foreground and background color indexes to use for the text and the background, respectively. These are also the colors used for drawing when no other explicit color specification is made.

A default ViewStyle is assumed by new views. A different ViewStyle can be specified at any time, from the right mouse button menu, or within code. We defined a ViewStyleManager which manages named ViewStyles. It also broadcasts the system default to scheduled views. It is accessible by dictionary protocol. A new ViewStyle for a view can also be selected from a menu of available ViewStyles. ViewStyles can be constructed and shared among applications through the ViewStyleManager.

Cooperating Palettes

We designed a number of ViewStyles that define PixelStyles and Palettes in a cooperating manner. We did this by allowing PixelStyles and Palettes to contain nil values. nil palette entries do not change colors previously defined in the hardware palette. Only non-nil values are loaded into the hardware palette. Therefore, if different non-nil palette entries are used for different views, activation of one view will not affect the coloring of other cooperating views or cause flashes of color when control passes between views. Cooperating ViewStyles generally consist of Palettes which each define different foreground and background colors, using different palette indexes. They also typically load the upper half of their Palettes with colors appropriate for color dithering.

To ensure that transient graphics appear in specific colors, regardless of the ViewStyle of the view from which they were executed, color values can also be explicitly declared and managed. ViewStyle, PixelStyle and Palette respond to the message install. Installing a view style installs a palette into the hardware palette. It also installs a pixel style. Therefore, installing a view style allows direct control of these globally accessible defaults. Each of these global attributes can also be installed individually as well.

Colors can also be changed for the duration of a block. Palette, PixelStyle and ViewStyle respond to the message showWhile:. The argument to showWhile: is a block. The block is executed after the appropriate values are installed. The Palette, PixelStyle or ViewStyle resource is restored to its prior values after the execution of the block.

For example, this code ensures that the 'black on ivory' view style is installed while the user is queried for a response:

```
(ViewStyleManager default at: 'black on ivory')
  showWhile:
    [FillInTheBlank request: 'File name?']
```

The query "File name?" is displayed with black text on an ivory background.

A background process that writes graphics to the display screen needs to observe a few rules to operate in a color environment. Background processes should not rely on the default PixelStyle or default ViewStyle. And they should not contend for the display palette. A background process should ensure that its desired Palette and PixelStyle are installed when it performs graphics. And if it intends to leave graphics on screen after it relinquishes control, it needs to construct a palette that is designed to coexist with other processes. Otherwise, the colors of any graphics it leaves on the display may change when the process relinquishes control.

The Smalltalk-80 development environment needed modification to work within a color framework. When a view is activated, its view style is installed. When the System Transcript prints a message, it must print in its view style, regardless of the view style of the currently active window. When windows are moved or framed, the screen must repaint uncovered portions in the correct colors. When the debugger comes up, it must appear in its own colors. Finally, when the user types <Ctrl-Shift-C>, the text typed to the emergency evaluator must be visible.

We modified a modest number of methods to make this work. Appropriate methods send messages either to directly install a Palette (as was the case of the emergency evaluator) or to direct a view to install its PixelStyle and/or Palette. Four one-line methods were added to View instance protocol which implemented the necessary functionality: installPixelStyle, installViewStyle, showPixelStyleWhile:, and showViewStyleWhile:.

Design Issues

Our implementation of color Smalltalk defined over forty new classes and added approximately 400,000 bytes to the size of the executable Smalltalk image. Clearly, our color implementation represents a significant increase in the complexity of our Smalltalk image. Reasonable questions to ask of our design are:

- How fast is it?
- How easy is it to port monochrome applications to it?
- How easy is it to port color Smalltalk to other hardware?

Performance

BitBlt operations on forms of depth one, or onscreen graphics, have been highly optimized in our implementation. The OVER BitBlt combination rule has been especially optimized. So have some other cases, such as halftoning. We carefully tuned the existing BitBlt calls in the Smalltalk development environment to exploit the optimized primitive BitBlt code. Consequently, graphics performance of the standard development environment in color Smalltalk, is as good as, or better than, our monochrome implementation.

Dithering is somewhat more expensive than drawing with a pure color, although it is still quite reasonable. For example, filling a colored form of extent 64 by 64 is 25% faster when using a halftone of depth one, than when using a dithered halftone. Using forms of other depths than depth one or the display depth requires using BitBlt simulation, which is also slower. Mixing forms of depth one and the depth of the display screen is supported by primitive code, and is therefore quite fast. Mixing forms of other depths requires a depth coercion algorithm, which is significantly slower.

Porting Monochrome Applications

Our color implementation has not yet been fully exploited by an extensive color application. So far we have made more sophisticated usage of color than our users. Most of our users are experienced in Smalltalk programming, not graphics. However, several users have managed to add color to existing monochrome applications in a day or two. Lines and backgrounds were colored, and multi-colored forms were incorporated into the application. For most applications, more color was unnecessary.

We encountered a few cases where monochrome applications broke when ported to color. One common problem arose when BitBlt combination rules other than OVER were used. These rules have no meaning in color, and lead to unexpected results. The problem can be solved in one of two ways. One can simply change the combination rule to OVER, replacing the destination with the effective source pixel. If that is undesirable, one must devise another mechanism to achieve a comparable graphics effect. For example, an area can be filled with a halftone of a specified color, or a colored line can be highlighted by changing its corresponding palette entry. How to add color effectively to a monochrome graphics application is not obvious in all cases.

Portability

It is difficult to design a system for portability without having first attempted ports to a variety of target machines. In general, classes that do not represent physical machine or architectural structures are portable. Classes that do encapsulate machine dependencies need rework. Our implementation can, with minor changes, accommodate differing hardware palette sizes or frame buffer depths. A single image and interpreter (with differing appropriate defaults and behaviors) support both a grayscale and a color model of a Tektronix workstation.

Our implementation represents Forms as pixel values which are mapped to palette entries. Color information is captured through associating a Palette with a Form pixel array. An architecture that

represents images as direct color specifications, for example, would require replacing the Form class definition and, quite possibly, rethinking the role of PaletteForms.

For efficiency we store pixels as a byte array of tightly packed pixels. The pixel ordering within the bytes matches the Motorola processor memory organization. An implementation for an Intel processor, or for a machine architecture based on bit planes rather than pixels would undoubtedly require reworking the Form pixel representation. Primitive BitBlt code most certainly is not portable across machine and processor architectures. Probably the most severe constraints to portability would be attempting to port color Smalltalk to a software or system architecture that did not support reasonably efficient color BitBlt primitive code implementations.

Conclusions

It is certain that Smalltalk graphics cannot stay where they are today; the world would pass Smalltalk by. There are many directions that Smalltalk graphics can go. Smalltalk implementations may require integration with emerging windowing and graphics standards such as X windows or NeWS™ [Gettys et. al. 1987, Sun Microsystems, Inc. 1987]. These systems define window and graphics facilities for applications. Each of these windowing and graphics systems defines an imaging operation that differs from monochrome or color Smalltalk BitBlt. Color support with different color BitBlt semantics has certainly been provided by other Smalltalk implementations [Digitalk87, Miranda87]. A Smalltalk implementation based on a particular windowing or graphics facility would not necessarily implement BitBlt graphics as part of the Smalltalk virtual machine; instead it would provide Smalltalk abstractions of services provided by the windowing system and call directly upon the window manager for those services.

Another possible direction for Smalltalk would be to base the Smalltalk graphics and development environment on a richer and potentially portable graphics imaging model, such as Postscript [Adobe Systems Inc. 1985].

We view our color Smalltalk implementation as an integrated system for developing interactive color graphics applications in Smalltalk. There is no standardization in the Smalltalk community for what constitutes a reasonable set of color graphics capabilities. We hope we have provided some insight into what constitutes a high level support for color and color graphics systems.

Acknowledgements

This paper is possible due to the incredible talent and energies of my teammates on the Tektronix color Smalltalk engineering team: Mike Miller, Roxie Rochat and Wes Hunter. Jocelyn Yu deserves special recognition for her BitBlt implementation. Merlin Miller was responsible for the hardware design and initial BitBlt design and implementation. His consultation and critique of our BitBlt semantics was invaluable. Lauren Wiener and Barbara Yates built our work into a usable system by documenting what color is and how to use it. Kim Rochat ensured the reliability of our design. And Allen Wirfs-Brock made BitBlt even faster when it needed to be.

References

[Adobe Systems Inc. 1985] Adobe Systems Incorporated, *PostScript Language Reference Manual*, Reading Mass: Addison-Wesley, 1985.

[Digitalk87] *Smalltalk/V EGA Color Extension Kit*, Digitalk, Inc., 1987.

[Foley and Van Dam1982] James D. Foley and Andries Van Dam, *Fundamentals of Interactive Computer Graphics*, Reading Mass: Addison-Wesley, 1982.

[Gettys et. al1987] Jim Gettys, Ron Newman and Robert W. Scheiffler, *Xlib — C Language X Interface Protocol* Version 11, Massachusetts Institute of Technology, Cambridge, Massachusetts, and Digital Equipment Corporation, Maynard, Massachusetts, 1987.

[Goldberg and Robson, 1983] Adele Goldberg and David Robson, *Smalltalk-80 The Language and its Implementation*, Reading Mass: Addison-Wesley, 1982.

[Miranda87] Eliot Miranda, "BrouHaHa-A Portable Smalltalk Interpreter", pp. 354-365, *Proceedings of OOPSLA '87*, vol. 22, no. 12, SIGPLAN, December 1987.

[Sun Microsystems, Inc.1987] Sun Microsystems, Inc., *NeWS Manual*, Mountain View, California, March 1987.

[Wirfs-Brock and Miller86] Rebecca J. Wirfs-Brock and Merlin R. Miller, "A Hardware Architecture Supporting Color Smalltalk", Poster Paper at OOPSLA 1986.

Appendix: Color BitBlt Semantics

Four Forms are involved in the most general BitBlt case: source, source clipping mask, halftone, and destination.

Either monochrome or color forms may be specified for source, halftone, and destination. Eight cases result. They are specified in the depth case methods in class BitBltSimulation. The following table introduces the notation used to describe the eight cases:

S source pixel value
H halftone pixel value
D destination pixel value
M source clipping mask pixel value
1 single bit per pixel (monochrome)
N multiple bits per pixel (color)
& boolean AND function (pixel & pixel)
r combination rule function (pixel r pixel)
f filter function (f (pixel))
s stencil function (s (pixel))
m source clipping mask function
(D m effective M)

The combination rule and the boolean AND (&) functions are binary operators that operate on the corresponding bits of two pixel values.

The filter and stencil functions use pixel transformation parameters that specify a variety of filtering, stenciling, and masking effects. They are stored in the pixel style derived from the halftone. The filter function transforms *n*-bit pixels into 1-bit pixels. To do this, it uses the instance variable filter of pixel style. The stencil function transforms 1-bit pixels into *n*-bit pixels. To do this, it uses the instance variables foreground and background of pixel style.

The eight cases are specified below. The name of each case is intended to suggest certain uses for the case. Other uses are certainly possible, however. In all but two cases (filled and stippled), the boolean AND function is applied to some version of the source and halftone. That intermediate result is termed the effective source. The combination rule function is then applied to the effective source and some version of the destination.

In the filled and stippled cases, the boolean AND function is applied to the source clipping mask and either the source or the halftone. That intermediate result is termed the effective source clipping mask. The combination rule function is applied directly to the destination and either the source or the halftone.

The source clipping mask must be of depth one and have the same extent as the source. If sourceClipMask is nil, then it is considered to be a black form (all ones) of depth one, and of the same extent as the source. A destination pixel may be modified only if the corresponding pixel in the effective source clipping mask has a value of one.

The pixel style foreground and background values are used for stenciling. The pixel style filter value is used for the filter function (stenciled, filtered halftone, filtered source, and filtered effective source cases). The plane mask value from the pixel style is used for all cases except monochrome; it is used for reading in conjunction with the filter cases and for writing when the destination is colored (cases stenciled, filled, stippled, and color mixed).

Case	S	H	D	Effective Operation	Name
1	1	1	1	((S & H) r D) m M	monochrome
2	1	1	N	s((S & H) r fD) m M	stenciled
3	1	N	1	((S & fH) r D) m M	filtered halftone
4	1	N	N	(H r D) m (S & M)	filled
5	N	1	1	((fS & H) r D) m M	filtered source
6	N	1	N	(S r D) m (H & M)	stippled
7	N	N	1	(f(S & H) r D) m M	filtered effective source
8	N	N	N	((S & H) r D) m M	color mixed

We defined BitBltSimulation, a subclass of BitBlt. It simulates pixel copying for class BitBlt. It is useful for debugging in case of a primitive failure with BitBlt. BitBltSimulation contains an executable specification of our color BitBlt imaging operation written almost entirely in Smalltalk code. BitBltSimulation also operates on Forms whose depths are not supported by primitive code. Primitive code implements BitBlt between depth one forms, forms whose depth matches the Display, and between forms of depth one and depths the same as the Display. BitBltSimulation was a very useful aid in validating primitive BitBlt code during development. We defined three new primitives in our BitBlt implementation: color BitBlt, a primitive that returns the value of a specified pixel for a form, and a primitive to set a specified pixel within a form. These pixel accessing primitives operate on forms of any depth and, incidentally, are used in BitBltSimulation.