SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 25(6), 637–655 (JUNE 1995)

Adding Graphics to a High-level Programming Language

CLINTON L. JEFFERY
Division of MCSS, The University of Texas, San Antonio, TX 78249, U.S.A.

AND

RALPH E. GRISWOLD AND GREGG M. TOWNSEND
Department of Computer Science, The University of Arizona, Tucson, AZ 85721, U.S.A.

SUMMARY

When graphics input/output capabilities are added to a programming language originally designed with a text stream input/output model, various design decisions affect the ease with which the graphics facilities are learned and used by applications programmers. In adding window system facilities to the Icon programming language, some design decisions were made very differently from the conventional wisdom, resulting in substantial benefits for programmers. In addition, some pre-existing Icon language features have proved to be useful in graphics programming.

KEY WORDS: graphics; window systems; Icon; programming language design

BACKGROUND

Input/output activity is a fundamental aspect of computing, especially with the trend toward interactive computing. It has increased in importance as successive generations of computers have become more interactive and as computer use has spread to non-technical persons. Early programming languages had input/output models based upon the crude batch processing available at the time. When those languages began to be used for programs with interactive input/output, adjustments to the languages were made, but it is very difficult to change an established language, and extending a language with entirely new programming models does not help the plight of those maintaining existing programs. Newer languages support interaction more naturally, since they were designed with that function in mind.

A similar process is now taking place for graphics, as interactive text-based models of input and output are being replaced by models that accommodate graphics and graphical user interfaces. Languages such as Smalltalk,[1] which were developed on systems where every user was equipped with a bitmapped-graphics display, provide a consistent programming model for graphics. Most popular languages, however, were *not* developed with an assumption that graphics input and output facilities are available. Because extending these languages is difficult, most languages have incorporated graphics capabilities by means of function libraries, rather than embed-

CCC 0038–0644/95/060637–19

Received 14 February 1994
Revised 13 December 1994

ded language support. Furthermore, these function libraries generally provide a programming model that is incompatible with and unrelated to the previous, text-oriented input/output facilities.

The end result of this approach is widely known: learning to program the widely used graphics and window systems is a difficult task; one that is different from and taught separately from the task of learning to program in a given language. In order to program for graphics, programmers must first discard what they already know about input/output coding and adopt a new way of thinking. It was in this context and as a reaction to these problems that the embedded graphics programming facilities discussed in this paper were conceived.

DESIGN CRITERIA AND OBJECTIVES

Icon's graphics facilities were designed with ease of programming as a high priority. This primary design criterion in Icon results in a drastic simplification of the graphics programming facilities compared with the C language graphics facilities used by the implementation: Xlib[2] and Presentation Manager.[3] Xlib consists of over 330 functions and several dozen new data types; Presentation Manager has closer to 500 functions. In Icon the graphics and window programming facilities are reduced to 47 functions and an extension of an existing type. The importance of simplifying the programming interface cannot be overestimated: conventional wisdom indicates that competence in programming a major window system interface in a standard language requires between six and twelve months of training, even for experienced programmers. The situation is even more severe when an application must include native support on multiple window systems.

The complexity of the graphics facilities in modern window systems is a direct reflection of their common ancestor: Smalltalk. Smalltalk pioneered this area with comprehensive support for graphics applications in an object-oriented language. Smalltalk provides dozens of new classes (types) for objects such as points, rectangles, images and common combinations, such as forms (images with a point offset specifying their origin). There are hundreds of methods to learn for the various graphics classes. Programmers can write graphics applications efficiently once they have adopted Smalltalk's mindset and invested sufficient time to learn the facilities provided, but the start-up cost is high. In contrast, adopting Icon has a low entry cost for programmers familiar with procedural languages, and the graphics facilities are accessible immediately and allow more sophisticated concepts to be learned gradually.

A second major objective in the design of Icon's graphics facilities was the integration of the graphics subsystem into the existing language in a consistent way. This objective raises several issues, the first of which is the difficulty with which the new input/output facilities are adopted in existing programs and by existing programmers. Some existing language systems, notably Borland C++,[4] have partially integrated a new graphics programming model with the earlier text model by means of libraries that emulate the standard text-oriented applications programming facilities under the window system. Borland's product is successful in allowing text programs to run in a window, and it does allow such programs to be enhanced with additional graphics windows. However, writing a Borland C text application using

standard I/O and console I/O functions is a completely different task from writing an application that deals with graphics systems features such as a locator device.

A third major objective was to leave the programmer in control of the program. Most graphics facilities mandate an entirely new programming model that is unrelated to existing input/output techniques. The classical and popular terminology for this model is event-driven programming, and its dogma is almost universally accepted. The basic tenet of event-driven programming is a good one. It states that at every instant the user, instead of the program, should be in charge of the interaction with the computer; in other words, the computer exists to take the orders from the user, not vice versa. When taken too far in language design, like most dogmas, the benefits of event-driven programming are achieved at the cost of an enforced program complexity that often is not necessary. In particular, it makes the job of adding graphics or the use of a locator device (such as a mouse) in text-oriented applications more difficult than is necessary.

A good example of this is Microsoft's Visual BASIC,[5] where the user interface coding has been completely replaced by a drawing program. The programmer, however, is no longer free to define the central control flow of the program, but instead writes callback procedures for each of the interface objects defined during the drawing of the program's user interface. A programmer who wants to port his Microsoft GW-BASIC[6] program to Visual BASIC is faced with a daunting task. In order to add Visual BASIC's controls, the central control flow of the program must be deleted and state information that was inherent in the control flow must be explicitly maintained via global variables, or the event-driven paradigm must be subverted in some way.

By contrast, in Icon the programmer is free to decide when and how much the event-driven programming paradigm should be used. Because the graphics facilities in Icon are integrated with the pre-existing text input/output model, programmers can write programs in the same way they always have. Adding graphics features such as locator input to a text-oriented application represents a minor enhancement rather than a redesign of program logic.

These three design criteria—simplification, integration and control—determine the general characteristics of Icon's graphics facilities, and result in several beneficial side-effects. For example, the substantial simplification of the graphics facilities results in increased ease of implementation; because of simplification there are fewer functions and features that must be implemented in order to make Icon run on a new window system. Although this article discusses Icon primarily in terms of its relationship to the X Window System graphics facilities in Xlib and the OS/2 Presentation Manager, other window system ports are under way. Comparisons between Icon and underlying graphics facilities such as Xlib and Presentation Manager apply to other window systems' graphics facilities to the extent that they are similar to Xlib or Presentation Manager; for example Microsoft Windows (and Windows NT) is structurally similar to Presentation Manager from the applications programmer's point of view.

Another example of graphics facilities that mandate event-driven programming can be found in the Tcl language's Tk toolkit.[7] A flexible set of user-interface capabilities is provided, but the facilities provided require the use of callback procedures to handle user input.

THE ICON PROGAMMING LANGUAGE

Icon is a high-level, imperative programming language with a large repertoire of operations on strings and structures. Some of Icon's characteristics have a significant effect on how the language's graphic facilities are cast. Conversely, some of Icon's features are particularly useful in graphics programming. This section briefly reviews the relevant features of Icon.

Expression evaluation

Although Icon is an imperative programming language, it has a sophisticated expression-evaluation mechanism that has a flavor of logic programming. In Icon, the evaluation of an expression can produce a value (*succeed*) or produce no value at all (*fail*). Unlike most imperative programming languages in which Boolean values are used to control program flow, Icon uses success or failure to control program flow. Thus, if an expression cannot perform a computation, it fails and that failure determines whether other computations are performed. For example,

```
text := read()
```

assigns the next line of input to text if there is one but fails on an end-of-file, in which case the assignment is not performed and the value of text is not changed.

Success and failure also control loops, as in

```
while text := read() do
  write(text)
```

which copies input to output. The loop terminates when read() fails.

Similarly, a comparison operation succeeds or fails depending on whether or not the specified relation holds. For example,

```
if count < 0 then write("negative count")
```

writes an advisory message if count is less than 0.

Some Icon expressions, called generators, produce a sequence of results if the context in which they are evaluated requires alternatives. For example, the function find(s1, s2) generates the positions at which s1 occurs as a substring in s2.

The iteration control structure causes a generator to produce all of its results in sequence. For example,

```
every i := find(s1, s2) do
  write(i)
```

writes all the positions at which s1 occurs as a substring in s2.

Alternative results also are produced by a generator if they are needed to produce the success of an enclosing expression. For example, in

```
if find(s1, s2) > bound then
  write("out of bounds")
```

```
else
  write("in bounds")
```

the control clause causes find(s1, s2) to produce successive results until one is greater than bound, in which case the expression in the then clause is evaluated, or until find(s1, s2) has no more alternatives, in which case the expression in the else clause is evaluated.

Types

Icon supports many different types of data, including integers, real numbers, strings and several kinds of structures.

In Icon, variables are not typed but values are. There are no type declarations and any variable can have a value of any type. Icon provides run-time type checking and coercion to ensure that the arguments of operations are of the correct types.

The types of values used in control structures are irrelevant; only success and failure are important. In the case control structure, in which an expression is selected depending on its value, the value can be of any type and different selectors can be of different types. An example is

```
case zero of {
  0: write("integer zero")
  0.0: write("real zero")
  "0": write("string zero")
  }
```

Structures

Structures in Icon are first-class values that are created during program execution. Storage management is automatic; space is allocated when a structure is created and garbage collection frees space used by structures that can no longer be accessed.

Icon supports records, lists, sets and tables. All of these structures can be heterogeneous; that is, they can contain values of different types.

Lists are one-dimensional arrays. A list can be created by specifying each value in it, as in

```
primaries := ["cyan", "yellow", "magenta"]
```

or by specifying the number of elements and giving an initial value for all elements, as in

```
points := list(500, 0)
```

which creates a list of 500 elements, all of which are zero initially.

Lists can be accessed by position, as in

```
poinsts[100] := 1
```

which sets the 100th element of points to 1.

Lists can also be accessed as stacks and queues. For example,

```
put(colors, "lavender")
```

adds the string "lavender" to the right end of colors, increasing its size by one. Similarly,

```
foreground := get(colors)
```

removes an element from the left end of colors and assigns it to foreground. If colors is empty (that is, it has no elements), get(colors) fails and the value of foreground is unchanged.

Tables in Icon provide associative look-up. They resemble lists, but they can be subscripted by values of any type. A table is created by

```
contexts := table()
```

which assigns an empty table (with no elements) to contexts. Subsequently, elements can be added to a table by assignment to subscripted references, as in

```
contexts["normal"] := "white"
contexts["warning"] := "red"
```

String scanning

One of the most interesting features of Icon is string scanning, a high-level pattern-matching facility. String scanning is motivated by the observation that many analysis operations are often performed on a single string while moving from place to place in it.

String scanning makes pattern matching simpler by providing a subject string that is the focus of attention for analysis operations for which no string need be specified explicitly. A position in the subject is maintained automatically, avoiding notational detail.

An example of string scanning is

```
text? {
  while tab(upto(&letters)) do
    write(tab(many(&letters)))
  }
```

The string text provides the subject of scanning, which is implicit in the body of the scanning expression that follows it. Scanning starts with the position at the beginning of the subject. The function upto(&letters) produces the first location in the subject at which a letter occurs. tab(upto(&letters)) moves the position to this location in the subject. In the do clause, many(&letters) produces the position of the last of a sequence of letters, and tab() moves to this position. It also produces the substring of the subject between the previous and new locations, thus matching a 'word', which is written.

Note that the functions in the scanning expression do not explicitly refer to the subject, thus simplifying their form. Similarly, there is no explicit reference to the position as scanning proceeds—it is changed by the scanning functions as desired portions of the subject are found and processed, but it is never necessary to know its numerical value.

The subject and position in string scanning have global scope. During the course of string scanning, they are available throughout the program, not just in the static scope of the scanning expression. Thus, for example, programmer-defined procedures can be called from within a scanning expression to perform more complicated kinds of analysis.

String scanning expressions can be nested statically and dynamically. The current subject and position are saved when a new scanning expression is initiated and restored when it is complete.

Although the subject and position are implicit and maintained automatically in a scanning expression, they are available as the values of the keywords &subject and &pos, respectively. Thus write(&pos) writes the current position in the subject. These keywords can be thought of as attributes of the environment in which string scanning takes place.

OVERVIEW OF THE GRAPHICS FACILITIES

Icon's graphics facilities provide a new data type, window, and operations on that new data type. There are several fundamental ways in which this type and its operations simplify the large number of types and functions needed for graphics programming and window management. This section presents key features of Icon's graphics facilities in comparison with the underlying native facilities for graphics programming. See Reference 8 for a complete description of Icon's graphics facilities.

Windows as terminals

The window data type is an extension of Icon's file data type. Windows may be substituted for files in the language's existing file input/output operations. In such usages, a window operates in a manner similar to a computer terminal when file input/output is performed. A file has only a small internal state (most notably the current offset or position at which input/output is taking place). In contrast, a window has a more substantial internal state, including a text cursor position (analogous to conventional terminal cursor position), the current window contents, as well as various font, color, and graphics style attributes that affect the appearance of output.

In addition to the window-as-terminal programming model, every window simultaneously supports a graphics programming model consisting of a two-dimensional array of pixels. There is no mode-switching between text and graphics. Graphics input/output does not affect the text model in any way, and vice versa, except that output in either model overwrites and obscures prior output in the same location within the window. Viewing windows as 'graphics terminals' is consistent with the model actually provided by window systems such as MGR[9] and 8½.[10] Icon implements this model as a higher-level abstraction on top of window systems that lack it, such as X.

A fundamental aspect of the window model is that window contents are *retained*

when a window is reduced to an icon or obscured by another window. There is no concept of window exposure in Icon; window repainting is handled automatically by the Icon implementation. Retained windows are essential in providing the programmer with the freedom to organize program control flow in a manner that is appropriate to the application instead of requiring that organization revolve around window system events.

As an example of the integration provided by the terminal model, consider a text application that interprets keystrokes h, j, k, and l as commands that move the cursor left, down, up, and right, respectively. With appropriate bounds checks to prevent the application from attempting to move off-screen, the Icon code for such a command loop is

```
cursor_row := cursor_col := 1
repeat {
  case getch() of {
    "h": if cursor_col > 1 then cursor_col -:= 1
    "j": if cursor_row < rows then cursor_row +:= 1
    "k": if cursor_row > 1 then cursor_row -:= 1
    "l": if cursor_row < cols then cursor_col +:= 1
    }
  # move cursor to (cursor_row, cursor_col)
  }
```

Adapting such a program to take locator input as an alternative to keystrokes is as simple as adding a clause to the case expression. When reading input from a window, values that represent locator activity are processed in a fashion similar to that of keystrokes. Where keystrokes are indicated by one-character strings, locator actions are indicated by special values designated by keywords. For example, the value &lpress indicates that the locator's left button was pressed. When processing input, the Icon keyword &row provides the text row of the locator at the time the input occurred and &col provides the corresponding column. With a clause to handle locator input, the text cursor can be made to jump directly to a row and column indicated by a locator button press with the code

```
cursor_row := cursor_col := 1
repeat {
  case getch(window) of {
    "h": if cursor_col > 1 then cursor_col -:= 1
    "j": if cursor_row < rows then cursor_row +:= 1
    "k": if cursor_row > 1 then cursor_row -:= 1
    "l": if cursor_col < cols then cursor_col +:= 1
    &lpress: {
      cursor_row := &row
      cursor_col := &col
      }
    }
  # move cursor to (cursor_row, cursor_col)
  }
```

Other small changes in a text application are required, such as opening a window when the program starts, and specifying the window instead of standard output in input/output operations. The point of this example is that control flow need not be rewritten in order to take advantage of window system resources in simple applications.

Encapsulation of complex features in windows

In comparing our graphics facilities to one of the native graphic systems they use, one obvious simplification is that a call to an Icon function typically results in many underlying graphic system function calls. Similarly, the underlying system objects used during window input/output operations, such as network connections, graphics attributes, and graphics context information are all packaged together into a single source-level window object. The programmer is free to ignore them and can expect reasonable default behavior. This approach makes simple applications easy to develop and allows gradual increases in sophistication and functionality as an application matures.

Encapsulating multiple system objects in the window abstraction and composing higher-level operations from multiple graphic system calls are techniques typical of higher-level toolkits and graphics languages. Icon makes no claim to uniqueness in this respect. Rather, Icon can be viewed as an extreme case of the use of these techniques: window system independence and ease of learning are achieved by implementing all window system operations in terms of normal language values such as strings and integers.

The first level of abstraction beneath the window model bears further examination, since it significantly affects programming style. A window is a *binding* of two components: a *canvas* and a *context* (Figure 1). A canvas is a two-dimensional array of pixels with associated attributes such as size, a physical manifestation on-screen (canvases may also reside off-screen), text cursor location, and a string label that

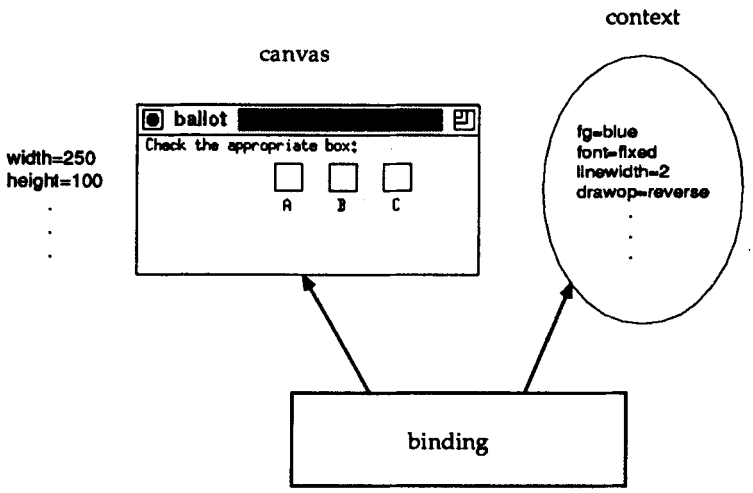

Figure 1. A binding

identifies the canvas for the user. A *context* consists of a set of attributes used in drawing operations such as the foreground and background colors, text font, and line width.

A canvas is an abstraction of one of the native system objects on which drawing operations are defined, typically called windows, bitmaps, and pixmaps. A context is an abstraction of a collection of the native system objects used in drawing, such as pens and brushes. By packaging a canvas and context together, Icon reduces the number of arguments required by window operations: a binding provides the network connection, window, font and drawing style arguments commonly needed by native system functions.

There are other arguments for the canvas and context abstractions. Some systems allow sets of graphics attributes to be set up in advance and used without having to specify each attribute in each drawing operation or perform repeated attribute changes between operations. This is an important performance consideration on systems that employ a client/server model, such as X.

The goal of simplifying the programming interface is achieved in Icon by allowing bindings to share canvases or contexts. Bindings that share a canvas allow multiple contexts to produce output on that canvas with a concise notation and high performance (Figure 2). Bindings that share a context allow a consistent set of graphics attributes to be applied to multiple canvases (Figure 3).

Graphics facilities

The graphics facilities provide the kind of generality and flexibility found in the rest of the Icon language. The basic functionality is provided by the system's native graphic routines to draw or fill objects such as lines, arcs, and polygons of various sorts. In Xlib, these operations are provided by two functions each, a singular version that takes coordinates (such as x and y) in-line, and a plural version that takes an array of structures containing the coordinates. In contrast, Icon's graphics

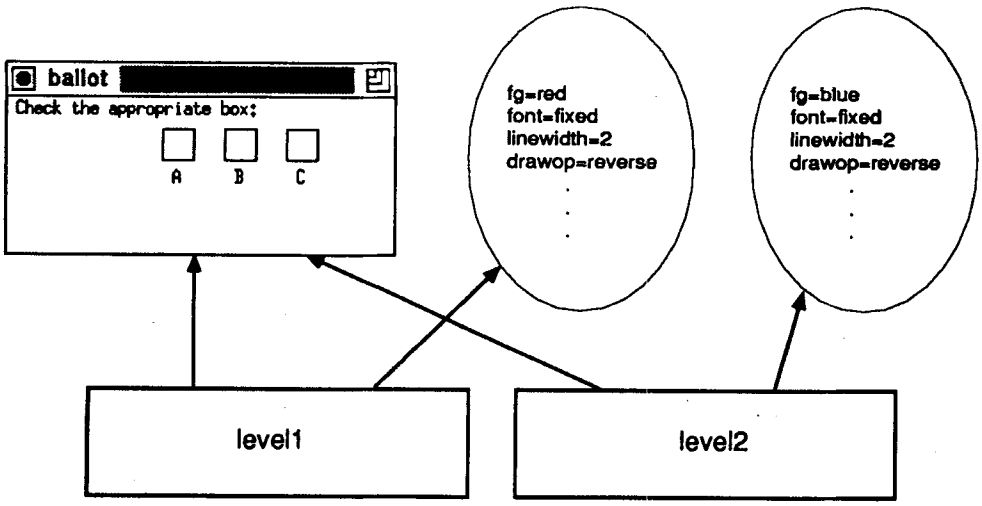

Figure 2. Bindings allow multiple contexts to draw on a canvas

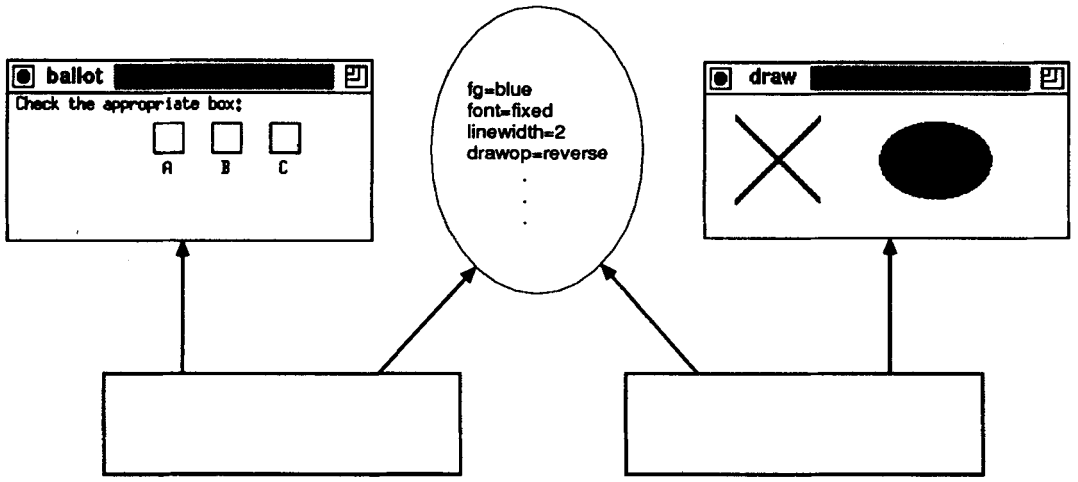

Figure 3. Bindings allow a context to draw on multiple canvases

functions all take an arbitrary number of arguments, and automatically convert their arguments to the appropriate type (such as integer pixel coordinates). Special types for graphics objects such as points and rectangles are not employed.

In addition to handling a variable number of objects, many arguments may be omitted and default to values appropriate for the graphics function in question. For example, a circle is obtained by drawing an arc at a specific location and with a specific width; the height of the arc defaults to the width and the starting angle and extent of the arc default to produce a complete circle. This kind of defaulting behavior provides a concise programming notation while minimizing the number of functions the programmer must learn. A further example is provided by the use of the default window, &window. This global variable provides a default window argument to the various graphics drawing functions. The net result of these cumulative details is that in order to draw a circle, the Icon programmer need only write

```
DrawArc(x, y, diameter)
```

instead of, for example, the corresponding Xlib (C language) call:

```
XDrawArc(display, window, context, x, y, width, height, angle, extent)
```

Because this simplification is achieved by the provision of default values, it does not reduce the capability of the function repertoire; it merely allows simple operations to be specified in a simple way and in a notation that is consistent with related but more complex operations.

Window attributes

Since an Icon window consists of a collection of underlying window system components, a mechanism is needed for manipulating these features. For a binding, Icon abstracts a complex set of internal structures into a single set of *attributes* and

associated *values*. Attributes and values are strings, and they are queried and assigned by means of the function WAttrib(). For example, the call

```
WAttrib("height")
```

returns the height of &window in pixels. The attribute may be changed by following it with an equal sign and a value:

```
WAttrib("height=300")
```

There are various shorthand notations in Icon for commonly-used attributes and combinations of attributes, but the basic simplicity of the attribute-value model provides a conceptual framework that is independent of the window system and is easily learned.

The set of attributes associated with a given binding is divided into two parts: attributes of the canvas and attributes of the context. The distinction is important because, as noted earlier, canvases may be bound (and subsequently drawn upon) by different contexts and similarly a given context may be bound to (and used to draw on) different windows. This underlying complexity in the window system is an example of the kind of capability that Icon hides from the novice without denying access to the capability for expert programmers writing sophisticated applications.

Color and font naming

Although many older programs written for a monochrome text-only terminal do not use colors and fonts, many sophisticated applications now use these attributes to produce high-quality output. Colors and fonts pose the most serious portability issues in the Icon graphics facilities. Icon's windowing capabilities can be built on top of almost all modern graphics-capable systems, but the variation in color capabilities and font support among platforms is large.

The problem of variation in color naming is addressed by the interpretation of a common set of color names within Icon. Names such as 'light blue' are converted into numeric RGB specifications which are passed to the window system. The color names are inspired by a system proposed earlier,[11] with certain additions.

Fonts pose an even greater problem than colors in that a portable application can make few assumptions about the fonts available on the system. Available fonts vary not only among different graphic systems, but also among different releases and different vendors' implementations of the X Window System. Icon can use whatever fonts are available on a given system, but it is the burden of the application writer to determine what fonts to use on the systems the program will run on, or to write code that works with user-selected fonts. In the interests of source code portability for typical applications, Icon provides four portable font names corresponding to the best available fixed- and proportional-width, serif and sans-serif system fonts. The portable names are typewriter, mono, serif, and sans and application programmers can expect them to be available in many type sizes.

Color and font resource management

Beyond the problem of specifying colors and fonts in the source program lies the problem of managing resources in use. Colors and fonts are allocated and deallocated by Icon without programmer attention beyond the specification of a color or font when it is used. This automatic management is both an outgrowth of the binding model and a natural extension of the automatic storage management provided by Icon for other data types.

The implementation of automatic color management is complicated by the fact that many platforms have only 16 or 256 colors available for all the windows on the screen. On such platforms, Icon's run-time system must free unused colors aggressively. Reclamation of colors no longer in use is a cumbersome and arcane task requiring examination of the contents of windows and pixmaps as well as the graphics contexts used to draw on them. Icon makes reasonable compromises to provide functionality with acceptable performance. Colors are shared with other applications when possible, and a given color is only requested once in an Icon application no matter how often it is referenced or whether it is used in multiple windows. When an application requires more colors than are available on the shared system colormap, a virtual colormap is allocated implicitly on those platforms that support it. If an application requests more colors than can be satisfied by any means, the request fails. The window system frees colors when a window is erased or closed. It is worth noting that the management of colors is not needed on some systems, notably with those with 'true color' hardware support and on window systems such as OS/2 that provide implicit dithering to generate unlimited colors.

Fonts are less troublesome than colors from the resource management point of view: They occupy space, but this generally is not a problem. Fonts are allocated by Icon as needed and never freed. Fonts are only loaded once and shared among all canvases and contexts on a given server; this minimal degree of resource management has worked well in practice.

ICON FEATURES USEFUL IN GRAPHICS PROGRAMS

In addition to the contributions and experience gained from the graphics facilities themselves, experience writing graphics programs with Icon has revealed some basic aspects of graphics programming that are handled well by features of Icon. Languages that possess similar features can gain similar benefits in the domain of graphics programming. The flexible argument handling discussed in the preceding section—sensible defaults, automatic type conversions, and variable-length argument lists—has proven useful in graphics as it has in Icon's other application domains.

Another Icon language feature that is useful in graphics is control structure heterogeneity. Icon control structures, most notably the case expression, allow clauses of any data type, and different clauses may be of different types in the same control structure. In the domain of graphics programming, this allows simpler code than homogeneity would require. Consider the problem of processing user input in a program such as a desktop publishing application. The program performs various actions depending on the nature of the user's input; ordinary text might be written to a window in some font, control keys might result in editing of existing text, while mouse activity might move the current focus of attention within the document.

The case expression is the natural multi-way selection construct in many program-

ming languages, but in processing window system input, it poses a problem. Input consists of different kinds of data; key presses are fundamentally different from mouse actions. A key press is naturally represented as a one-character string with the ASCII value of the key pressed, but such a representation is not so appropriate for mouse activity. Various forms of control and function keys present a similar representation problem. In order to solve this problem, most graphics programming facilities characterize user input as a variant record of some sort with a common field indicating the nature of the input.

This complex representation of input activity imposes an additional layer of logic on applications in order to get the natural form for manipulating that input in the program. At the very least, the programmer must remember more functions and when to use them. Similar complexity is introduced when the graphics facilities use a variant record to encode all forms of user input in a consistent manner: The programmer has more types to learn and more fields to recall.

In contrast, Icon represents key presses as strings and locator actions as integers. Related information such as the locator position at the time of the input is delivered via associated keywords, as shown in the earlier locator input example. Since Icon's case expression allows values of any type in the case-selectors, no variant record is needed and little or no mental effort is involved in decoding input events or processing mixed key presses and mouse events.

As an example of the effects of this language feature, consider a loop that reads events until the user presses either the left mouse button or the escape key. In real applications there can be many of these sorts of case branches. In Icon, the loop looks like this:

```
repeat case Event() of {
  "\e" | &lpress : return
  # . . . other events . . .
  }
```

By comparison, the corresponding code in Xlib looks like this

```
for(;;) {
  XNextEvent(display, &event);
  switch (event.type) {
    case KeyPress: {
      char s[5];
      KeySym ks;
      int i = XLookupString(&event, 2, 5, &ks, NULL);
      if ((i == 1) && (*s =='\033')) return;
      /* . . . handle other keystrokes . . . */
      break;
      }
    case ButtonPress: {
      if (event.button == Button1) return;
      }
    /*. . . other events . . . */
    }
  }
```

Presentation Manager code for the same task is even more complex. Control is split into multiple locations and event data is transmitted in several separately-passed 32-bit parameter values (msg, mp1, and mp2):

```
/* . . . in a main procedure . . . */
while (WinGetMsg(thisprocess, &msg, 0, 0))
  WinDispatchMsg(thisprocess, &msg)

/* . . . in a window callback procedure */
  switch (msg) {
    case WM_CHAR: {
      keyflags = (unsigned short) mp1;
      if (keyflags & KC_KEYUP)
        if (TransKeyMsg(keyflags, mp2, &keyval)) {
          if (keyval == '\033') return;
          /* . . . handle other keystrokes . . . */
          }
      break;
      }
    case WM_BUTTON1DOWN:
      return;
    }
```

Because Icon's table data type provides associative look-up for keys of any type, tables are useful in mapping values from an application's problem domain onto the visual characteristics used to depict visual representations. An earlier example showed a table that maps program values "normal" and "warning" onto color names "white" and "red", respectively. Tables can also directly map program values to bindings whose drawing context attributes are set to the corresponding colors as discussed earlier. The Icon function Clone(win, *attrs*) creates a new context and binds it to an existing canvas, producing a new value of type window that has independent color and drawing attributes. For example, if the application domain is a set of terrain types, then a table constructed with the code

```
color := table(Clone(win, "fg=black"))
color["ocean"] := Clone(win, "fg=dark blue")
color["lake"] := Clone(win, "fg=light blue")
color["desert"] := Clone(win, "fg=tan")
color["wood"] := Clone(win, "fg=light green")
color["forest"] := Clone(win, "fg=dark green")
```

enables the application to draw on win with simple Icon calls such as

```
FillPolygon(color[terrain], x1, y1, . . .)
```

In addition to color codings, tables of bindings support codings that use other context attributes such as texture patterns, line widths and styles, or fonts. The preceding example could just as easily represent terrains with both a color *and* a pattern, improving the application's appearance and supporting color-impaired users. Tables

of bindings also make it simple to supply alternative codings, such as gray scales for documentation in place of colors normally used in the application.

Another Icon language feature that is useful in graphics programming is the built-in list data type. Icon lists are heterogeneous structures that allow both deque (stack and queue) and random (positional) access. In most window system facilities, window events that have not yet been processed are kept on an *event queue*. In Icon, the event queue is in fact an Icon list. Instead of adding a whole set of functions for inspecting or inserting elements on the queue, the Icon graphics facilities use existing built-in operations.

Icon is not unique in allowing applications to produce virtual events. Because Icon uses simple types such as strings instead of complex structures to represent user input, virtual events are easy to create. Although motivated by the desire to make event decoding tasks convenient, the simplification naturally makes event encoding tasks more convenient as well.

FUTURE DIRECTIONS

Experience with Icon has shown that many programs that use graphics can be written quite conveniently using conventional programming techniques. Not all graphics programs are fundamentally different from conventional programs, nor are they necessarily more complex. However, programming details often present a problem.

The present design of Icon's graphics facilities leaves room for improvements, especially in the area of the notation used to manipulate the attributes that affect window output behavior, such as the color and line width used during drawing. One further improvement would be to eliminate the reference to attributes through function calls in favor of one in which attributes are manipulated as fields of windows, in the style of record manipulation. This would not only simplify code related to attributes, but it would allow numeric attributes to be expressed in their natural form, instead of being encoded as strings. Thus, instead of

```
WAttrib("height=" || incr * 300)
```

the following could be used

```
&window.height := incr * 300
```

A more radical approach is needed to raise the level of graphics programming to the level of string and structure operations in Icon. Icon's string scanning facility, in which an environment for pattern matching is maintained automatically, suggests a possibility. *Rendering expressions*, in which a window is the current focus of attention would allow the attributes of the subject window to be manipulated without the need to specify the window. A rendering expression might look like this:

```
window ? {
   . . .
  &bg := "black"
```

```
&font := "fixed"
        . . .
}
```

Of course, there are substantial differences between scanning environments, in which the only 'attributes' are &subject and &pos, and rendering environments, in which there could be dozens of attributes. The really interesting challenge is the design of rendering functions analogous to the string-analysis functions of string scanning.

Since such dynamic environments would couple data elements closely with code that manipulates those elements, these techniques are similar in certain ways to those encountered in object-oriented programming. There is a conciseness afforded by associating a focus of attention in the data with the sections of code primarily concerned with manipulating that data. Unlike object-oriented methodologies however, dynamic environments do not enforce encapsulation but instead leave the question of how rendering environment code fits into the rest of the application up to the programmer. This freedom provides its biggest gains in smaller applications where the object-oriented notations are cumbersome.

CONCLUSION

Icon uses a novel approach in the addition of graphics capabilities to a programming language with a conventional text-oriented input/output model. Icon succeeds in providing a smooth integration with pre-existing text operations and programming models. Experience with Icon has shown that simple windowing programs can be written in 10 to 100 lines, instead of the hundreds or thousands of lines required by many application program interfaces. Despite the limitations of a simplified programming model, Icon affords a concise notation with which to implement a wide range of graphic, window-based applications.

A large library of programs and procedures written in Icon, including an interface tool kit and an interface builder, already exists and provides a resource of reusable code for future applications. An example of an application written using these facilities is given in the appendix to this paper.

Icon's graphics facilities at present run under Presentation Manager in OS/2, X on UNIX and VMS platforms, and implementations for Windows, NT, and Macintosh platforms are under way. Icon and its library are in the public domain. Icon is available by anonymous FTP to ftp.cs.arizona.edu; cd/icon and get READ.ME for navigation instructions.

The approach used here to adding graphics to a programming language is not limited to Icon; it can be used for any high-level programming language. Some aspects of the design described here, such as the handling of colors and fonts, can be applied in any language. In general, however, it is important to exercise care to design the graphics facilities so that they fit naturally into the language and are integrated with other features rather than being an *ad hoc* extension. In Icon, integration was facilitated by using capabilities such as generators and functions with an arbitrary number of arguments. Other languages offer different possibilities for the language designer.

ACKNOWLEDGEMENTS

Darren Merrill ported the facilities to OS/2 and provided the impetus to change the graphic facilities so that they were not window-system specific.

This work was supported in part by the U.S. National Science Foundation under Grant CCR-8713690 and a grant from the AT&T Research Foundation.

APPENDIX: AN ANIMATED KALEIDOSCOPE

This appendix shows a graphics application written in Icon. The application displays an animated kaleidoscope in which randomly sized and placed circles are drawn in eight symmetrical positions (Figure 4). The number of circles builds up until a specified density is reached, at which point the oldest set of circles is erased before a new set is drawn.

Using sliders, the user can adjust the speed at which the display is drawn, the density of circles, and their minimum and maximum radii. The File menu at the top allows the user to save the current display in an image file or quit the application. The pause button stops the animation, and the reset button clears the display to start over. The radio buttons at the bottom allow the user to choose between disks (filled circles) and rings (outlined circles).

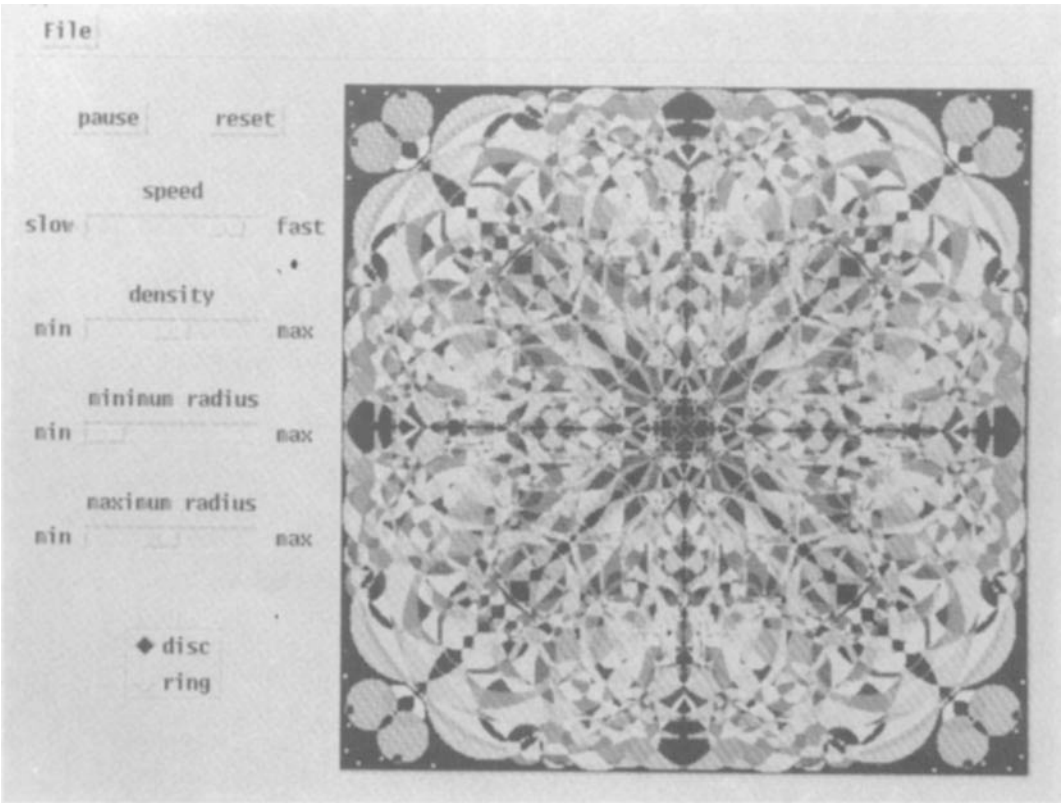

Figure 4. A kaleidoscope application

The interface tools are provided by a library of procedures written in Icon.[12] The application itself was constructed using a direct-manipulation interface builder, also written in Icon.[13]

The application itself consists of 182 lines of Icon code, including initialization, a procedure to produce the display, and callback procedures for the interface tools.

REFERENCES

1. Adele Goldberg and David Robson, *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, Massachusetts, 1983.
2. Adrian Nye (ed.), *Xlib Reference Manual*, O'Reilly & Associates, Inc., Sebastopol, California, 1988.
3. Charles Petzold, *Programming the OS/2 Presentation Manager*, Microsoft Press, Redmond, WA, 1989.
4. Borland, *Borland C++ Version 3.1 Programmer's Guide*, Borland International, Inc., Scotts Valley, CA, 1992.
5. Kenyon Brown, *Programmer's Introduction to Visual BASIC*, Sybex, San Francisco, 1992.
6. Microsoft, *Microsoft GW-BASIC User's Guide and User's Reference*, Microsoft Corporation, Redmond, WA, 1986.
7. John K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, Reading, Massachusetts, 1994.
8. Clinton L. Jeffery and Gregg M. Townsend, 'X-Icon: an experimental Icon windows interface version 8.10', *Technical Report 93–9*, Department of Computer Science, University of Arizona, April 1993.
9. Stephen A. Uhler, 'MGR—C language application interface', *Technical Report*, Bell Communications Research, July 1988.
10. Rob Pike, '8½, the Plan 9 window system', *USENIX Summer '91 Conference*, June 1991, pp. 257–265.
11. Toby Berk, Lee Brownstein and Arie Kaufman, 'A new color-naming system for graphics languages', *IEEE Computer Graphics & Applications*, May 1982, pp. 37–44.
12. Jon Lipp, 'Window interface tools for Version 9.0 of Icon', *Technical Report IPD259*, Department of Computer Science, University of Arizona, June 1994.
13. Mary Cameron and Gregg Townsend, 'VIB: a visual interface builder for Icon', *Technical Report IPD 258a*, Department of Computer Science, University of Arizona, August 1994.