Dynamically Displaying a Pascal Program In Color

John F. Cigas

Department of Mathematics & Computer Science
Ithaca College
Ithaca, NY 14850
Bitnet:cigas@icunix

Abstract

This paper describes a method of using color to display the actual structure of a Pascal program on a color monitor. This enhancement not only increases a programmer's understanding of the code, but also aids in detecting common structural errors. The paper identifies several structures deserving of color and the properties that must be adhered to when assigning colors to these structures. A simple coloring scheme illustrates this discussion. The last section describes enhancements and directions for future research.

I. Introduction

Color monitors capable of displaying text as well as graphics are now available for personal computers and workstations. Many software packages currently use color to distinguish between different functions or modes of operation. In this paper, we propose to extend the use of color to the display of computer programs.

Previously, several programming environments have incorporated the formatting of a program into its display on the screen. Pascal-I [Cichelli80] is an interactive Pascal system which reformatted the source code to proper indentation levels during every compilation. Other tools, such as the Cornell Program Synthesizer [Teitelbaum81] and SED (Syntax-EDitor) [Allison83] allow for multiple displays of a program, but do not reformat the program text on the display. This is to avoid disturbing the programmer's sense of location when viewing the program. It was felt that the programmer should have ultimate control of such formatting decisions.

One context-sensitive editor explicitly displays the program structure with an alphanumeric structure index beside each line of program text [Atkinson81]. From this display, the user can determine the exact contextual location of each program statement. However, the indexes take up many columns of space on the screen and require the user to perform an explicit mental transformation from the key numbers to the program structure.

We propose a similar method of displaying a computer program, except that we use color instead of alphanumeric indexes to distinguish between each of the control structures. This allows more space on the screen for the program and eliminates the transformation from indexes to program structure.

Our proposed display complements the traditional format of indenting sections of code enclosed in a program unit of a loop, procedure, or decision branch, but does not replace it. It differs from such formatting conventions in that the colors are not user-supplied; they are generated by the displaying program based on the parsed structure of the program. Since the colors are generated by our tool, not the user, the programmer sees what the compiler is doing, as opposed to what the programmer *thinks* the compiler is doing. This removes one level of uncertainty by displaying the exact structure of the program.

Enhancing the readability of a program is not novel. Program formatters and pretty printers have been around for years. However, they require the extra step of generating hard copy, then reviewing the program. Our method has the advantage of being an interactive tool used at the terminal. This paper explores the topics of what structures to color, what properties should apply to the colors, and how to choose the colors to satisfy the aforementioned properties. It only briefly attempts to justify the usefulness of color.

In the following section, we briefly give an example of how color can aid in detecting structural errors in a Pascal program. Such errors are usually caused by typing mistakes, not algorithmic misconceptions. The next section discusses some of the important properties needed to color a program. The fourth section presents one simple method of assigning colors to sections of a program and discusses how this conforms to the described properties. The final section describes the improvements that can be made on this simple method and indicates the direction of future research.

II. Common Structural Programming Errors

There are many common programming errors that color will help to detect. These include a missing comment terminator, missing END statement, an ELSE paired with the wrong IF, and a missing BEGIN-END around a block of code. This section illustrates the cases of a missing right comment terminator, "}", and a block not enclosed by BEGIN-END.

A missing comment terminator is a typographical error made by most programmers at one time or another. It is not a difficult error to correct, but it may be difficult to find. Consider the program in Figure 1. It is not a long program, and at first glance may seem correct. However, the Turbo Pascal 4.0

compiler refuses to accept the program, putting the cursor on the keyword UNTIL and displaying the message "Error 113: Error in statement". Of course, the error is not in the marked statement; it is not even in the statement immediately preceding it. The error is caused by the missing right brace "}" on line 8. Given the brevity of the program, it is not difficult to detect the error eventually, but it requires scanning the whole program. For a longer program, this becomes a significantly harder task.

```
 1  PROGRAM NoComment;
 2  VAR
 3    Count : INTEGER;
 4  BEGIN
 5   { This is a header comment. It
 6     contains many lines of text. It
 7     may appear anywhere in the program
 8     Alas, it has no closing bracket
 9    Write('Enter number of iterations: ');
10    Readln(Count);
11    REPEAT  { Execute "count" times }
12       Process;
13       Count := Count - 1;
14    UNTIL Count = 0;
15    Writeln('Program completed');
16  END.
```

Figure 1.

However, if the comments were displayed in a different color than the executable statements, the error would be immediately obvious. Such is the case with Figure 2. (The colored text is printed as underlined text.) In this case, it is very easy to see that several lines of seemingly executable code are in fact swallowed up as part of a comment.

```
 1  PROGRAM NoComment;
 2  VAR
 3    Count : INTEGER;
 4  BEGIN
 5   { This is a header comment. It
 6     contains many lines of text. It
 7     may appear anywhere in the program
 8     Alas, it has no closing bracket
 9    Write('Enter number of iterations: ');
10    Readln(Count);
11    REPEAT  { Execute "count" times }
12       Process;
13       Count := Count - 1;
14    UNTIL Count = 0;
15    Writeln('Program completed');
16  END.
```

Figure 2.

Another programming problem easily detected by color is a block not enclosed by BEGIN-END. This commonly occurs when adding a statement to a loop containing a single statement. Figure 3 shows a loop intended to assign zero to all the elements in two arrays. This loop may appear to be correct because the programmer has indented both assignment statements the same amount. Since this fragment is syntactically correct, the error will not appear until the program is executed and its output checked. However, if all statements within a loop are the same color, Figure 4, the programmer can easily see that the second assignment statement is not part of the loop.

```
FOR Student := 1 TO MaxStudent DO
   Grade[Student] := 0;
   Average[Student] := 0;
```

Figure 3.

```
FOR Student := 1 TO MaxStudent DO
   Grade[Student] := 0;
   Average[Student] := 0;
```

Figure 4.

These are only two examples of the usefulness of dynamically coloring a program on the display. Other examples are omitted here in the interest of brevity. In the next section we discuss what parts of a program should be colored and some criteria for choosing each of the colors.

III. What to Color in a Program

In the previous section, we presented an example of how color can be used to reflect the structure of a Pascal program. Now, we shall discuss the parts of a program to be colored. These are important considerations, because one must be judicious in the use of color. Using too many colors only results in a cluttered display that doesn't convey any meaning to the viewer. Furthermore, even with high resolution monitors capable of displaying many thousands of colors at a time, there is a limit to the number of colors that a viewer can distinguish and remember at any one time. Our application requires a large contrast between colors in order to be functional.

As seen in Figure 2, it is useful to color comments differently from executable statements. Additionally, among executable statements, there should be a distinction between each of the different control structures. The three necessary control structures are *iteration, repetition*, and *decision*. A fourth structure, *subroutine*, is also useful and will be considered as well. Of these four, there is no need to color iterative statements separately, since they do not reflect a change in the control flow. The other three structures do affect control flow and should be colored distinctly. We examine each one separately.

Repetition (loop). In Pascal, there are three types of loops: WHILE, FOR, and REPEAT. The differences among them are not important from a coloring point of view. It is only important to be able to discern that statements are inside the body of a loop. Given this distinction, the programmer can easily read the code to determine the type of the loop. Therefore, all types of loops can be the same color.

Subroutine. Coloring Pascal's two types of subroutines, PROCEDURE and FUNCTION, follows the same reasoning as for loops. Given that they are colored differently from other structures, the programmer can determine the type of subroutine from context. Therefore, both procedures and functions can be the same color.

Decision. There are two types of decision statements, IF-THEN-ELSE and CASE. The CASE statement could be colored, but because of its inherent structure and well-defined scope, we choose not to consider it further. The IF statement, though, presents some other challenges. It must be colored, but there is a choice on how to color the ELSE branch. The ELSE may be the same color as the IF, to show the pairing, or it may a different color, to indicate the difference between the two branches. Arguments can be made for both options. For the remainder of this paper, we will choose to pair the ELSE with its corresponding IF in the same color. This is an arbitrary choice and requires further investigation.

One final area to receive color is the declaration section of a program or subroutine. It is useful to differentiate this section with a distinct color, not so much to detect errors, but to separate it from the executable statements in a program. We also find it visually appealing to do this, since it breaks up a large block of one color into two sections of different colors.

IV. Coloring Criteria

Having decided what to color, there are some properties to be aware of when deciding what colors to use. These are as follows:

Consistency. Each control structure should have its own unique color. That color must not be used for any other control structure. This allows the programmer to recognize structures by color as well as by keyword.

Similarity. While all color applications require consistency, displaying a program adds an additional constraint when displaying nested structures. If all loops are the same color, as required by the consistency criteria, then there is no way to distinguish between the inner and outer loops. Therefore, the color of a structure must change depending on its level of nesting. However, there is still a need for some consistency, so the color change can not be random. The solution is to change the color a small amount, enough to indicate a change (see distinction) but not enough to overlap the color space of another structure.

Distinction. Distinction means that one must be able to tell two different structures apart by sensing two different colors. Even though a monitor is capable of displaying many fine shades of color, there must be a fairly large contrast between two adjacent control structures, such as nested loops. Note that it is easy to provide contrast between six predetermined colors, but it becomes more difficult when taken in context with the principle of similarity, which adjusts the color of a structure based on its level of nesting.

V. A Simple Coloring Scheme

In this section, we describe a simple method for assigning colors to the different components of the program. We first define some terms, then we present our coloring scheme. Finally, we discuss some of the improvements that can be made to the coloring scheme.

The standard color monitor is composed of three phosphors for every pixel. These phosphors approximate the three colors red, green, and blue (RGB). Each phosphor has a range of intensities, mapped to the range 0..1, and operates independently of the other two phosphors in the same pixel. Therefore, to describe the actual color of a pixel, we need an ordered triplet representing the values of each phosphor. For example, (1,0,0) represents a bright red, (0,.5,0) a medium green, and (1,0,1) a bright magenta (red+blue). Equal intensities of all three components represent neutral gray colors, ranging from black (0,0,0) to white (1,1,1). Colors described this way are in the RGB color space. There are other systems for describing a color, and we shall mention some of these later. The RGB color space is not the easiest one to manipulate for color transformation, but conceptually it is the simplest to understand.

Since any two colors must be distinguishable, for each phosphor there is some minimum amount of change in its intensity that must occur for a noticeable change in color. In this example, we shall assume that this difference, d, is the same for each primary, and is constant over the whole range of intensities. Given M intensities in the range 0..1, there are M/d distinct colors that can be displayed by changing a single phosphor value. For this discussion, we pick $M=64$ and $d=16$, giving 4 distinguishable intensities per phosphor.

The structures of the program to be colored are: the main program declaration and its executable statements, procedure/function declarations and their executable statements, the declaration section of each block, loops, IF-THEN-ELSE statements (with the ELSE the same color as the matching IF), and comments. We will assign these six elements to six vertices in the RGB color space as follows:

program	- red	(1,0,0)
subroutine	- magenta	(1,0,1)
loop	- blue	(0,0,1)
declaration	- cyan	(0,1,1)
decision	- green	(0,1,0)
comment	- yellow	(1,1,0)
background	- black	(0,0,0)

Coloring will take place one of two ways. If the structure detected is different from its enclosing structure, the color used will be the initial color for that type of structure (as defined above). However, if the structure is the same as the enclosing structure, then the color is shifted one division (d) "down" on the color table. For example, the basic color of a loop is blue (0,0,1). Given a nested loop, the color of the inner loop would shift in the direction of cyan. This is accomplished by increasing the amount of green by d/M, or 0.25, giving the color (0,.25,1). This allows for up to four nested structures of the same type, more than enough for our simple examples.

This scheme satisfies the properties of consistency, similarity, and distinction. Coloring is consistent because the same nesting of structures results in the same colors being displayed. This applies both to different sections of the same program and to separate programs. This coloring satisfies the property of similarity, since nested structures are represented by only a small change in the displayed color, thus maintaining a close relationship with the previous color. The coloring is also distinct, since every color change is either to a vertex (very discernible) or an increase of at least d/M, which is the minimum discernible change. However, there is a limit to this distinction. The scheme only allows for four levels of nesting. Deeper nesting might be handled by repeating the sequence of colors for a structure, though this allows the same color to apply to different contexts.

The method of assigning colors to vertices along the RGB cube of colors and the simple step function means that any color

displayed is composed of at most two of the three RGB primaries. This appears to be a substantial restriction, reducing the number of available colors from M^3 to $3M^2$ and limiting the number of distinct steps that are achievable for coloring nested structures. However, adding a third primary does not necessarily increase the distinction on the display. Equal amounts of all three primaries result in a neutral color. As the third primary is added, the displayed color becomes less saturated and progresses toward gray. This is not a desirable situation, since it is easier to distinguish purer colors from less saturated ones.

VI. Future Work

Much work still needs to be done to refine the coloring of a program. Most of this centers around mapping colors to program structures and changing color based on the level of nesting. Some areas of interest are:

Change in color space. The RGB color space is easy to conceptualize, but difficult to work with [Hall89]. It is much easier to compute colors based on other systems, then transform them to RGB coordinates for display. The most advantageous system is the CIEXYZ system. It is important for two reasons. First, it describes colors independently from actual RGB values. Knowing CIEXYZ values for the individual phosphors in any monitor allows the user to build transformation matrices to and from the monitor's RGB space. Secondly, the CIEXYZ coordinates are easily converted into the uniform color space coordinates of the CIELUV system [Robertson77]. In this system, the distinction between two colors can be approximated by a straight line between the coordinates of the two colors. This allows for a more accurate computation of nested colors than just using a constant d. Employing the CIELUV difference metric allows for tertiary colors in the color display, without degenerating to a screenfull of gray images.

Non-uniform color assignment. Three of the six structures to which we are adding color are never nested - programs, comments, or declaration sections. This means that there will only be one color displayed for each of these structures. Consequently, to have more color available for the other structures, the initial allocation of colors to structures does not have to be uniformly spaced. Use of the CIELUV system greatly facilitates this allocation.

Foreground and background colors. The method described in this paper displays various foreground colors on a constant black background. It is possible to use a lighter background, and thus increase the perceived saturation of the displayed colors [Silverstein87]. It may also be useful to combine changes in both text and background colors. Care must be taken here, since many background colors are irritating to the viewer.

Completely unique colors. One of the problems with the current coloring system is that nested structures do not always possess the same colors. The only solution to this problem is to base coloring decisions on the global state of the program. Mathematically, this is fairly simple, but initial attempts resulted in a display full of drab, hard to distinguish colors. There is also the consideration that a user cannot discern more than a dozen colors at one time, and that generating a unique color for each configuration would be more confusing than helpful. This is the most difficult and interesting area of future work.

VII. Conclusion

We have described a method for displaying a Pascal program using color to indicate the structure of the program. This aids in detecting certain structural coding errors and in enhancing a programmer's overall understanding of the code. We have detailed various structures that should be colored and the color-related properties that must be adhered to when choosing colors. We have presented a simple example of a coloring scheme that satisfies most of the coloring properties. Finally, we have discussed the deficiencies of the simple scheme, outlined possible improvements, and indicated the direction of future research in the area.

References

[Allison83]
L. Allison, "Syntax Directed Program Editing",*Software Practice and Experience*,13, p.453-465, (1983).

[Cichelli80]
R.J. Cichelli,"Pascal-I - Interactive, Conversational Pascal-S",*SIGPLAN Notices*,15(1), p.34-44, (1980).

[Hall89]
R. Hall, *Illumination and Color in Computer Generated Imagery*, New York: Springer-Verlag, 1989.

[Robertson77]
A.R. Robertson, "The CIE 1976 Color-Difference Formulae", *Color Research and Application*, 2(1), p.7-11, (Spring 1977).

[Silverstein87]
L.D. Silverstein, "Human Factors for Color Display Systems: Concepts, Methods, and Research", *Color and the Computer*, Academic Press, (1987).

[Teitelbaum81]
T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: A Syntax Directed Programming Environment", *CACM*, 24(9), 563-573, (1981).