2-1994

An Algorithmic Palette Tool

Gary R. Greenfield

An Algorithmic Palette Tool

Gary R. Greenfield
Department of Mathematics and Computer Science
University of Richmond
Richmond, Virginia 23173
email: grg5l@mathcs.urich.edu

February, 1994

TR-94-02

1 Introduction.

Color — in particular, red, green, and blue (RGB) color — in graphics systems in usually controlled either directly by allotting to each pixel a certain number of bits to specify each of the red, green, and blue values for that pixel, or indirectly by allotting to each pixel a certain number of bits to serve as an *index* into a lookup table (LUT) which stores these RGB values. Thus when one speaks of "full 24 bit color" one refers to a system that stores per pixel eight bits each for red, green, and blue with the eight bits used to determine the percentage of that "primary" color that will be mixed to produce the color displayed at that pixel. In this manner one arrives at the oft quoted phrase about a color system allowing one to choose colors from a palette of

$$256 \times 256 \times 256 = 16,777,216$$

colors or, stated more succinctly, to choose from a palette of more than 16 million colors. Although occasionally one encounters extended 36 bit direct color systems, full 24 bit color is the sine qua non for color systems that are descended from the direct display eight-color systems where only one bit was devoted to each of the RGB values. This meant only zero or one hundred percent of each primary color could be mixed and gave rise to the infamous red, green, blue, cyan, yellow, magenta, white and black color palette.

The indirect method using LUTs is the one we shall be concerned with. It still allows one to choose from more than 16 million colors — because the LUT can store 24 bit RGB values — but the number of colors that can be displayed *simultaneously* is limited by the size of the table. Ordinary workstations with "eight bit color" allow one to display simultaneously $2^8 = 256$ colors constituting the palette that has been established for the application at hand. This report describes an algorithmic tool for LUT palettes, which when coupled with the user's aesthetic criteria, is found to be useful for *discovering* creative and ingenious palettes for visualization and artistic purposes.

The conceptual approach to using color in direct versus indirect displays is quite different. A full 24 bit color display supports and promotes the traditional mode of image creation: tools can help select, mix, and shade a color, and (via further tools?) that color can then be applied to selected portions of the image. But indirect color display has side effects. Altering an existing color in a selected portion of the image by a *substitution* in the LUT which will modify its hue, saturation and/or value must, by conse-

quence, affect that color's occurrence at every pixel throughout the image. This is both a boon and a bust. For example, it supports the primitive animation technique of erasing and drawing objects and regions by masking and unmasking their LUT entries.

A few hours spent wandering through the software and hardware vendor exhibition of a recent SIGGRAPH conference yielded demonstrations of various palette tools and widgets designed for use with LUTs which seemed for the most part to be based on one of the following three principles:

1. **Reassignment of the RGB percentages.** To illustrate the idea, one might imagine modifying an image so that every entry in the LUT where red is currently mixed at 40% is changed so that it is now mixed at 70%. To design a widget to accomplish this task one can use side-by-side graphs comparing the default linear percentages for the 256 possible red values with their reassignments. (See Figure 1.) Further, one can implement user control of reassignment by allowing the user to trace the reassignment curve or manipulate the control points of splines governing such a curve. An advantage to reassignment is that the LUT does not have to be modified, only reinterpreted.

2. **Loading the table with RGB values obtained from RGB percentage curves.** This is a very natural way to work with the LUT as it supports the RGB color mixing theory that color experts are well versed in. For its implementation, curves for each of the RGB values use one axis for the LUT entry and the other for the percentage. (See Figure 2.) Of course the curves should be aligned or simultaneously displayed in order to gauge the effect, but once more curve tracing and curve fitting can augment user control. Additionally, we have seen effective use made of expansion and compression. That is, a user can gain the full contrast of green in a brief table segment while simultaneously controlling for subtle differences in red. (See Figure 3.) A drawback would seem to be the emphasis on continuous, or piecewise continuous curves.

3. **Permutation of table entries.** This technique is another favorite used in special effects. Typically one circularly shifts the table entries or transposes large segments of the table. Both operations are especially conducive to control by scrolling, slider, or button widgets.

Our algorithmic tool follows the model of RGB percentage curves, but now the control of these curves is though algorithms that indirectly, and

more abstractly, create, evolve, and modify such curves. To fully explain our methods we must first introduce the topic "mutating expressions." This is done in Section Two. In Section Three we document the user-interface problems we dealt with, and finally in Section Four discuss conclusions and suggest ideas for future exploration. Before commencing with the technical details however, we wish to emphasize the nature of the "colorization" problem that led to the conception and development of our methods.

An image to be used in conjunction with an LUT display has two largely independent attributes. First, the *topological* characteristics of the LUT indices with respect to their distribution among the pixels. Second, the visual or aesthetic characteristics inferred from the entries (*i.e.*, palette) the LUT stores. Naively, we are merely saying that an image is a paint-by-numbers canvas. Its topological attribute is the the organization of the numbers on the canvas, and its visual attribute is the color scheme of the paints associated to the numbers.

The independent and sometimes conflicting nature of the topological and visual attributes is well-known. We first confronted and exploited the interplay between the contrast properties of the index distribution and the aesthetics of the palette in the algorithmic art tools we described in [2]. The subject has also received much recent attention by Scientific Visualization proponents because of the misleading *scientific* conclusions that might be caused by improper colorization of data [3]. And certainly artists are aware of the distinction between contrast by value (our indexes) and hue (our color). Witness the advice of Patricia Lambert in her how-to book Controlling Color [1, page 32]:

> Although value is only one of the four properties inherent in every color, it may be used independently. Therefore a design may *first be executed in value* before it is translated into color — an excellent habit to develop.

2 Mutating Expressions.

Our specification of RGB percentage curves will rely on functions of the form $f(u)$, in one independent variable u, with domain and range I, the unit interval. Given an LUT of size s with indices $0, 1, \ldots, s-1$ we would use f for say the red component by storing at entry i, $0 \leq i < s$, a red percentage of $f(i/s)$. In fact, most RGB scales use values between 0 and

255, so the table entry for red is the one associated with the nearest integer to $255 \times f(i/s)$.

To create, modify, and manage RGB-curves we use our own version and implementation of *mutating* or *evolving* expressions as introduced by Karl Sims [4]. Our functions are expressions that can be stored symbolically as trees, and can be evaluated at the independent variable u to yield a real number in the unit interval. The strength and power of using trees for functions relies on being able to adapt intrinsic tree algorithms for building and modifying trees. The end result is a user-controlled genetic algorithm to search for expressions that will provide palettes that are often unexpected, unimagined, unanticipated, and unplanned. That is, palettes that have aesthetic merit but that a person without traditional artistic training would be hard pressed to develop.

We need a more precise description. The leaves are our expression trees will contain either the symbol u, denoting the independent variable, or the symbol c, denoting a constant. Internal nodes of the tree will be functions in one or two variables and therefore internal nodes will have one or two branches respectively. This means we must provide a set of building block functions for the internal nodes with the property that both their arguments and their results must be restricted to lie between zero and one. To this end we defined *normalized* functions in one variable

nsin, ncos, nexp, nlog, nsqt, nsqr, ncub, nnot

and in two variables

nadd, nmul, nmod, nmin, nmax, npwr, nand, nvee, ncir.

For example, the graph of say the normalized cosine function *ncos* would show the classic cosine curve compressed and shifted so that it fits in the unit square. Similarly, the graph in the unit cube of three-space of *ncir* would reveal concentric circles whose projections onto the plane are centered at the point $(1/2, 1/2, 0)$. These projected circles are "lifted" so that they assume z-values starting at zero for the circle of radius zero and increasing as they radiate outward so that the circle whose projection passes through the vertices of the unit square is lifted to a z-value of one. The function *nand* is a normalized bitwise-and operator. Together with the two variable *nmod* and *npwr* functions they introduce fractal elements into the evolved and mutated expressions. In this report we choose not to rigorously define all our building blocks.

To add some measure of variation, and to assist in fine tuning expression trees, every node additionally stores two real numbers — a coefficient a and a constant b. The real numbers are used to modulate the value of the function value that is first calculated at that node. Formally, the function value r is an intermediate result which is modulated to produce the final node result

$$e = (a * r + b) \bmod 1$$

which will now serve as the argument value to the function of a parent node. (The function value of a leaf with symbol c is zero.) Our implementation always requires the constants b to lie between zero and one, but allows the coefficients a to be between zero and two because this promotes periodicity, a feature which was found to be useful in evolving palettes.

EXAMPLE 2.1. Since we found it easiest to "read" expressions when the coefficients and constants were printed in *front* of the functions for the nodes, the expression tree printed as

$$0.05\ 0.38\ nmin(0.54\ 0.09\ nsin(0.42\ 0.52\ u),\ 1.22\ 0.40\ nsqt(0.04\ 0.63\ c))$$

would correspond to the tree of Figure 4, and would be evaluated for LUT index i by setting using $u = i/s$, and calculating

$$(0.05 * nmin((0.54 * nsin((0.42 * u + 0.52) \bmod 1) + 0.09) \bmod 1,$$
$$(1.22 * nsqt(.63) + 0.40) \bmod 1) + 0.38) \bmod 1.$$

3 The User Interface.

Our user interface is written using the public domain Simple Raster Graphics Package (SRGP) written by David Frederic Sklar. We believe this package promotes reasonably rapid prototyping. The window we display for developing our palettes is shown in Figure 5. (The figure is unfortunately in black and white, but it will serve for descriptive purposes.) The upper left rectangle in Figure 5 is for displaying the image that one is currently colorizing. But thanks to the vagaries of the X window system we are able to colorize across windows because the window manager will perform a context switch of color maps as the mouse cursor is moved from window to window.

The rectangle to the immediate right of the one used for the image is a radio-button widget to allow the user to select either the red (RR), green (GG), blue (BB) or all (AA) three of the RGB-curves to create or modify. The rectangle immediately below the image rectangle gives a histogram indicating the proportion of pixels numbered by each LUT index — one understands the horizontal axis to be labeled left to right with the LUT indices — and it superimposes the current RGB-curves in their respective colors on the histogram. The histogram itself is scaled so that the index that is used most frequently by the image receives a bar which just reaches the top of the rectangle, while all other bars are proportional to this largest one. If an image concentrates its index distribution, the superimposed RGB-curves are suppressed at indices that have no bearing on the image. In any event, the wide rectangle at the bottom displays the current palette by drawing a vertical bar of the properly mixed color for each LUT index according to the horizontal LUT index scale of the histogram and RGB-curves rectangle. On the menu at the far right of Figure 5, the buttons labeled

Slide	Perturb
Feather	Differ
Vary	Evolve
Extend	Mutate
Drift	Birth

are the user controls for the RGB-curves. The RGB-expressions are displayed in printed form (using the format described above in Example 2.1) in the text window where the palette tool was launched from. The effect of any of the menu actions above on the RGB-expressions is echoed by scrolling the updated RGB-expressions in the launch window.

Regarding the individual buttons, selecting <Birth> initializes a curve by generating a "small"' random expression tree. The principal means of complexifying an expression is accomplished using the <Evolve>, <Extend> and <Mutate> buttons. <Evolve>, as one might expect, invokes an algorithm that in addition to giving every node a small chance of having its building block function altered, gives every leaf node a small chance of evolving to an internal node, and every one-variable function node a small chance of evolving to a two-variable function node. <Extend> is similar, but it only affects leaf nodes, thereby preserving the existing evolved internal structure of the expression tree. <Mutate> tries to emulate what might happen if an error is made in reproducing an expression. It does this by clipping a subtree from a copy of the expression and reinserting it in place of another

subtree of the expression. The other buttons implement milder modifications to expressions that could make sense if the expressions were viewed as genetic material from a population of organisms. Thus <Differ> alters the building block function in a single randomly selected node of the expression, and <Vary> does the same for a small percentage of the leaves of the expression. The remaining buttons focus on the constants a and b stored at each node and are used to foster "genetic variation." <Drift> imperceptibly shifts the constants at every node by a minute amount. <Perturb> also adjusts the constants at every node, but the fidgeting is weighted so that it becomes more severe as the depth of the node increases. Finally, <Feather> and <Slide> were especially designed for the colorization problem because they affect parameters of the expression that are particularly germane to palette construction. <Feather> fidgets only with the constants of leaves, while <Slide> fidgets only with the constant a of the root node.

The palettes we have worked with to date use $s = 180$ colors. We could use up to $s = 256$ colors, but the reason we chose 180 reminds us of an anecdote about buying hams: A daughter, as she had learned to do from her mother, always asked the butcher to cut off the end of the ham. When this behavior was brought to her attention, she conferred with her mother only to learn the reason for doing so was to make the ham fit in the pot! Our parallel anecdote stems from using a public domain palette editor authored by Gordon Cameron of Edinburgh University. The defaults for this editor are 180 colors, and we created some palette files using this editor that we transported to our application. Since those original palettes are still indispensable to us, we stick with $s = 180$. It is still true that with palettes of this size only one palette can be displayed at a time. Thus we were led to save expressions in circular buffers which operate independently for red, green, and blue. Frequently when a palette had several features we were trying to improve upon we found that we wanted to undo a modification and restore the original, thus the <UNDO> button was added to back up the pointers to these buffers. Also, because we often encountered promising palettes that we wanted to set aside for safekeeping, we added <PUSH> and <POP> buttons for moving them to and from a stack. In this manner, we could save several candidates on the stack, unstack them to the buffers, and then roll back through the buffers to make decisions on whether to save them to disk using the <ARCHIVE> button, restack them, or attempt to further develop their traits and characteristics.

The button <InPic> reads in the next available image from a file created especially for that purpose, and the button <NxtPal> cycles through

the RGB palettes from our image making software (see [2]) that are in the file format adopted by Cameron's palette editor, and that were used when creating the images we are trying to re-colorize. This explains the need for a <SwiPal> button to toggle between the original palette that was used to create the image, and the alternative palette we are creating using RGB-expressions. Of course all archiving, pushing, popping, and reading operations are echoed in the launch window.

EXAMPLE 3.1. The following image to be colorized is titled "The Entrance." It is also an expression tree, one that was created using our gb-software [2].

```
>P 0.77 0.81 ncir(0.71 0.43 nmax(0.73 0.50 ncir(0.69 0.80
ncir(0.73 0.39 nmax(0.81 0.65 v 0.21 0.10 v) 0.72 0.98 u)
0.86 0.64 v) 0.75 0.65 c) 0.12 0.09 u)
```

It was archived in the above form by the palette tool along with the following deceptively simple RGB-expressions we evolved for it. They provide a palette that added an outstanding color scheme.

```
>R 1.09 0.43 nmin(1.61 0.81 u 0.44 0.23 u)
>G 1.25 0.07 u
>B 1.19 0.36 nmod(0.80 0.76 c 1.25 0.31 c)
```

4 Conclusions and Future Inquiries.

We have been pleased with the initial experiments we conducted using our algorithmic palette tool, but there is still considerable room for improvement. In Figure 6, we have collected some sample RGB-curves that we created. (Again, black and white printing is a drawback, as only the red and blue curves could be reproduced.) It is gratifying to be able to verify from this figure that periodicity, fractal elements, and the modulo one "wrapping" are being drawn upon for palette creation. Especially telling is the piecewise phenomena exhibited by many of these curves, a feature which one hardly ever sees in connection with traditional palette tools.

The greatest drawback we have encountered so far is related to curve selection. First there is the problem of deciding at what point during palette development to concentrate on an *individual* red, green, or blue expression rather than *all* three at once. Second, it is easy to "forget" that at times the buttons are now affecting only a single expression. Since the <UNDO>

button is keyed to curve selection, popping from the stack and rolling back through the buffer(s) can destroy palettes in these situations. This problem can be partly remedied by synchronizing the buffers *i.e.*, *copying* idle expressions after each selection when only one expression is active. We are still at loss as to how best to remind the user that only a single expression is being operated upon though. The idea of suppressing the graphs of idle curves is contraindicated by the fact that colors are a mix of RGB values.

The design decision to superimpose RGB-curves on the histogram display is of questionable value. We fell victim to our own poor planning. Since we program so that windows can be resized, it became difficult for us to easily redesign the display so that the histogram, RGB-curves, and palette were aligned and displayed in the available space. We are also not sure how to best design the display in order to convey the information about which fragments of the RGB-curves are having a significant impact on the colorizing. Since many of our images use a surprisingly small number of LUT indices, this is an important question. We found suppression of the unused portions of the curves irritating, but found equally irritating buttons that modified expressions and produced drastic palette changes yet had little impact on the image because the key LUT indices were not adversely affected.

Continuing along these lines, when an image relies on just a few LUT indices, and these indices are also close together in the LUT table, it can take an inordinately long time to discover a palette that is sensitive to these indices. In such cases trying to modify palettes also proved frustrating. Many buttons seemed to have no effect at all. Our original default palettes had lots of sharp contrasts to overcome this problem and to help analyze and refine these kinds of images, evolved palettes have no specialized tools for this purpose. It is beyond our ability at this stage to incorporate methods for splicing palettes (an obvious attempt at a compromise?) since both displaying splices and managing data structures for splicing seem inconsistent with the expression tree approach.

Perhaps a stack is not the best choice for a data structure to use to temporarily set aside palettes. It would seem that a table that allows the user to direct palettes to and from the buffers would be more desirable. A table would probably be even more useful if there was an acceptable way to iconify the palette at each entry. One possibility is a table where each entry has three fields: STORE, RETRIEVE, and DISPLAY. STORE, would store the push of the active palette, RETRIEVE would pop onto the active palette, and DISPLAY would echo the saved palette expressions and give a pop-up visual readout.

The fact that RGB-curves are calculated using only one independent variable seems to limit their scope unnecessarily. However, we have drawn a blank when it comes to suggesting other independent variables for which it would make sense to try and place them under user *control.* Clearly, perfect user control is not our objective, but then again neither is perfect randomness!

An idea that we feel must definitely be explored is the concept of a hands-on visual display of the expression trees in tree-form. We are certain that if a user could be provided visual feedback about the effects of altering explicit subtrees or nodes in expressions, and had tools that would support this greater intimacy, the paradigm of mutating expressions would become more exciting and powerful. Therefore our long-term goal is to integrate improved palette tools, image tools (after [2]), and expression tree-editing tools into one unified package.

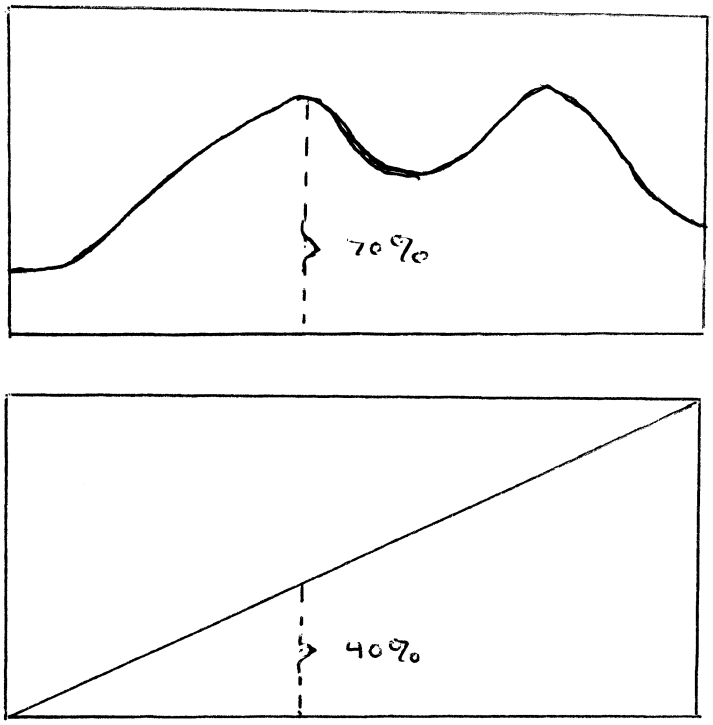
References

[1] P. Lambert, *Controlling Color: A Practical Introduction for Designers and Artists*, Design Press, New York, NY, 1991.

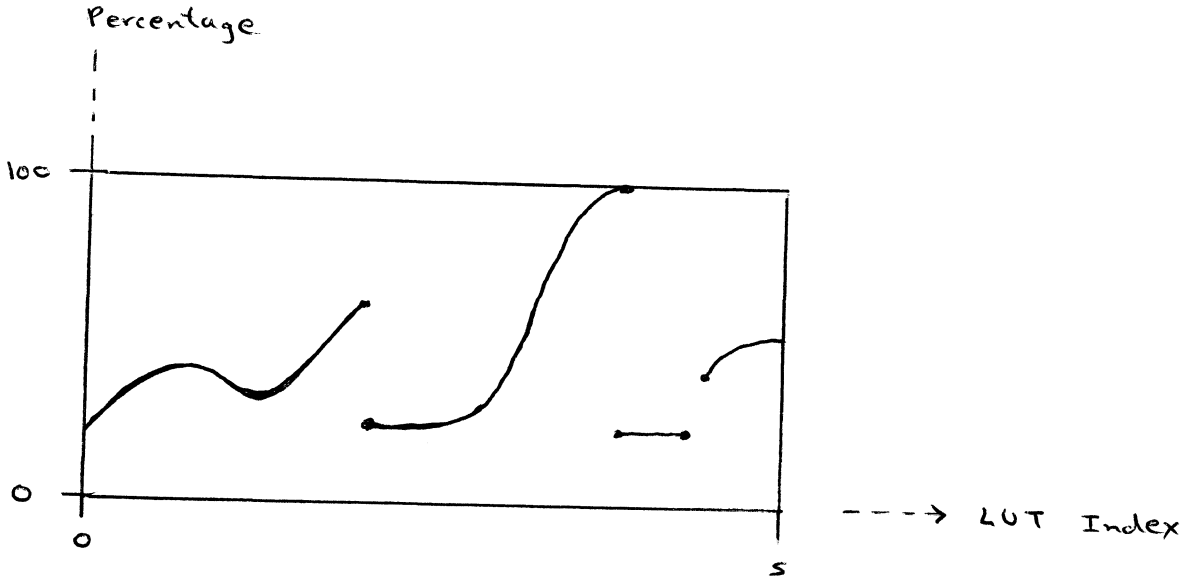
[2] G. Greenfield, Graphical evolution and computer art, preprint.

[3] N. Gershon, How to lie and confuse with visualization, *IEEE Computer Graphics and its Applications*, January 1993, 102–103.

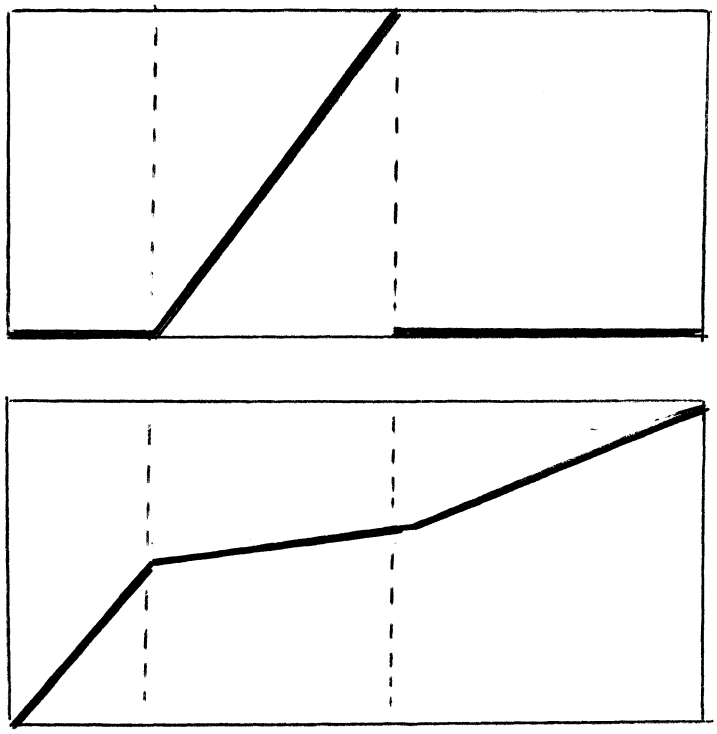
[4] K. Sims, Artificial evolution for computer graphics, *Computer Graphics*, **25** (1991), 319–328.



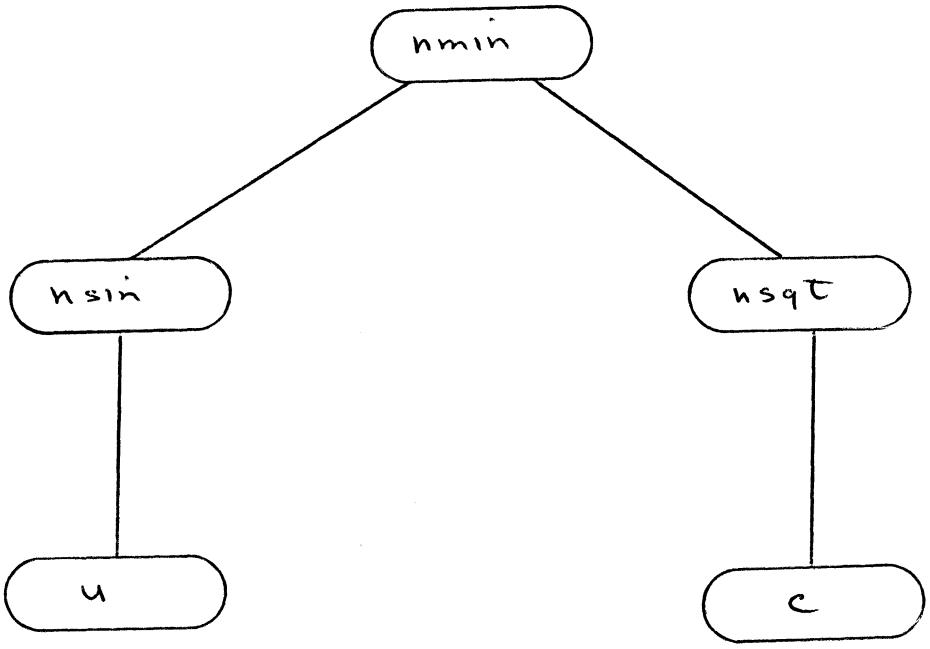
1. Reassignment Curve



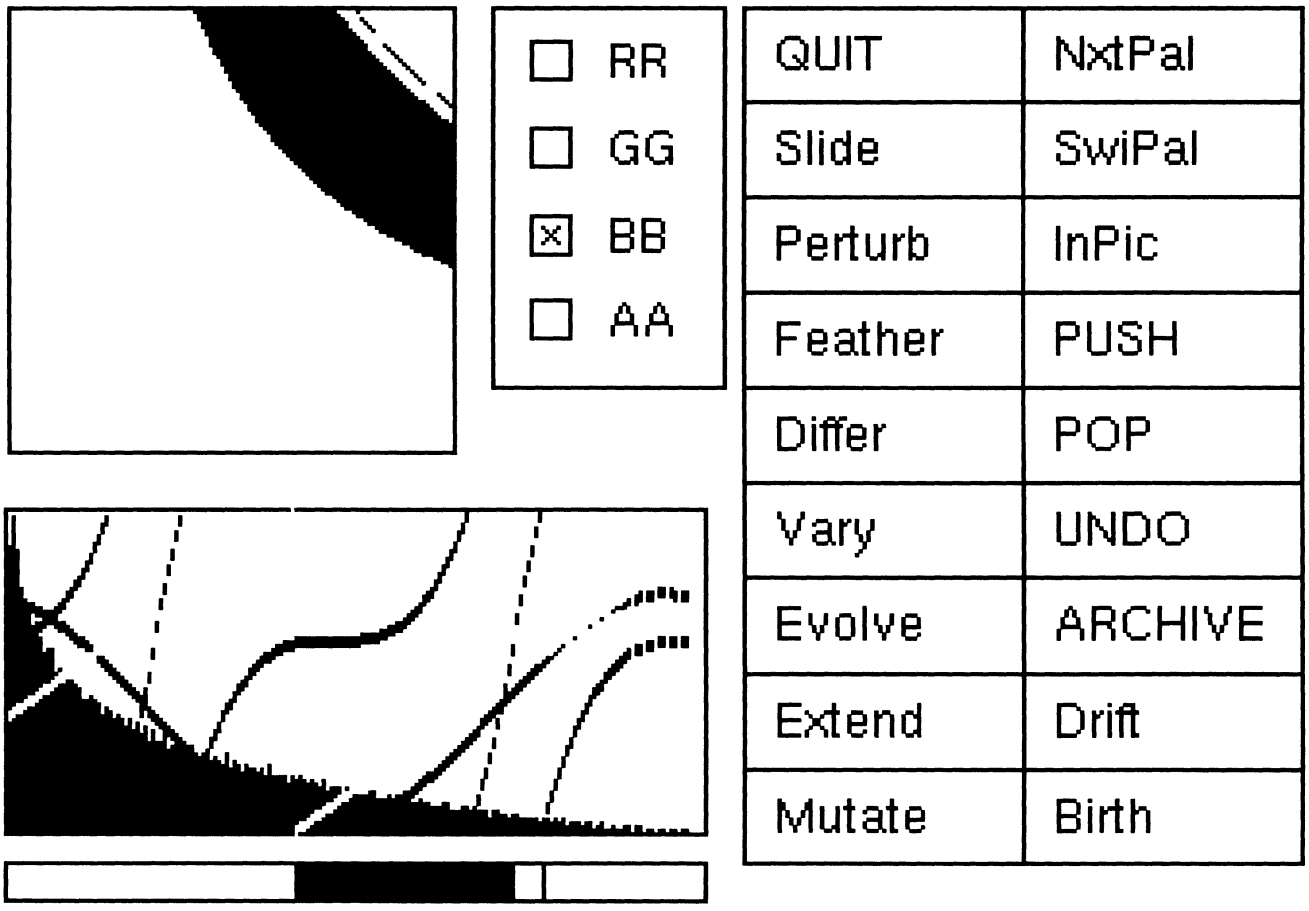
2. Percentage Curve



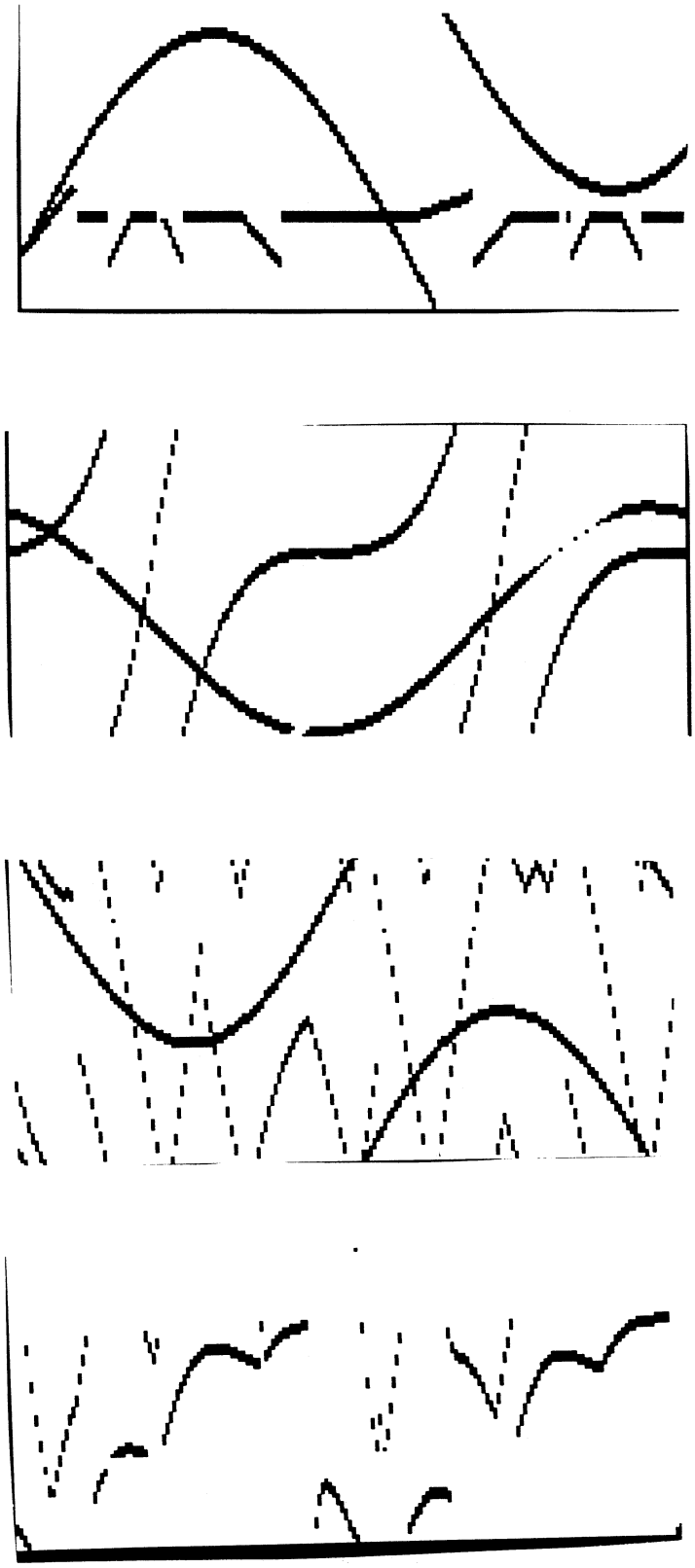
3. Compression and Expansion Curves



4. Expression Tree



5. Palette Window



6. Sample (Red and Blue) Expression Curves