DESIGN and EXPERIENCE with a GENERALIZED RASTER TOOLKIT

Alan W. Paeth and Kellogg S. Booth

Computer Graphics Laboratory, Department of Computer Science
University of Waterloo, Waterloo, Ontario, Canada N2L 3G1
Tel: (519) 888-4534, E-Mail: AWPaeth%watCGL@Waterloo.CSNet

ABSTRACT

Raster manipulation software is often viewed as an *ad hoc* means to fine-tune the appearance of digital images, or as a means to reformat them to conform to specific hardware requirements. A universally accepted, machine readable, device-independent specification of a raster image is seldom employed. This stands in contrast to the variety of "standards" for higher-level scene representation. We define a general raster "type", which unifies the design of a toolkit of raster-based software. Operations performed by the tools are closed in the sense that operators map objects having the raster type onto new objects having the raster type. This closure encourages a synthesis of function by allowing composition of operators. Sequences of these operators are surprisingly powerful and have wide application.

RÉSUMÉ

Les logiciels de manipulation d'images "raster" sont souvent considérés comme un moyen *ad hoc* d'améliorer l'apparence d'images digitales, ou comme un moyen de les modifier de façon à ce qu'elles se conforment à un appareil spécifique. La représentation universelle d'une image "raster", ne dépendant pas d'une machine particulière, est rarement utilisée; ce qui contraste avec le grand nombre de normes qui existent pour représenter des images de plus haut niveau. Nous définissons un type "raster" qui permet la création d'une série d'outils opérant sur celui-ci. Les outils en question forment un ensemble fermé dans le sens qu'ils opèrent sur des images de type "raster" pour produire des images de type "raster". Cette fermeture permet la création de fonctions par la simple juxtaposition d'opérateurs plus simple. Ces compositions de fonctions se révèlent étonnament puissantes et ont un vaste domaine d'applications.

Keywords: *bitmap, digital compositing, imaging, raster.*

INTRODUCTION

The ultimate goal of any software system should be the creation of a harmonious set of tools in which each tool embodies a conceptually simple operation. This is true for the case of raster image manipulation, but such a set is not in widespread use. To have generic utility, each tool must operate on an abstract raster type. For instance, a "cropping" tool should trim rasters regardless of their dimension or pixel attributes. Additionally, the tool's output should be, in all cases, a valid raster file so that tools may be composed arbitrarily.

To achieve this, we define a universal file format and implement general raster access routines. With these, the creation and coding of each new tool is greatly simplied, and the proliferation of disposable software can be alleviated. This scenario is also a boon to the user: generic tools imply a simple conceptual model. In some cases, they even suggest new ways of "plumbing" together raster operators. This approach is appealing in an academic/research environment, where creative experimentation is encouraged, but where software maintenance remains on a tight budget.

This paper discusses the design and implementation of a comprehensive raster manipulation system, based on the raster file format, that has been operational for over a year and is the mainstay of raster-based activities within the Computer Graphics Laboratory at the University of Waterloo. In that time, it has completely subsumed the various *ad hoc* raster file formats previously in use and has provided a unifying framework for new research.

OVERVIEW

The toolkit contains programs to support abstract operations (rotation and scaling), as well as interfaces to a number of hardware devices and software systems. These include I/O tools for Adage/Ikonas and Raster Technologies frame buffers, the Apple/Macintosh, and Versatec and Imagen hardcopy printers. In this usual setting, tools model the UNIX text/filter paradigm, whereby the output of any one tool may be piped

directly to the input of the next. This is particularly important when intermediate raster files may be quite large.

The tools are written in standard C, use no assembly code or specialized C packages (such as *yacc*), and have been ported successfully to machines with different word length and severe compiler restrictions. Conditional compiler code is used to represent the specifics of byte order. This allows the system to maintain both a uniform presentation of data for the low-level tool builder and an identical external representation (as a byte stream) for disk files.

Most users are not tool writers, but use the raster tools freely in a conceptual fashion. An artist working on the Macintosh might print bitmap files on the Imagen laser printer, or use them as picture input for the comprehensive Orcatech-based *Palette* [[Higg85]] colour painting system. Here the user may disregard the different pixel precisions, (lack of) colour, or machine word-lengths, all of which differ for each hardware system. Because the file format has been designed carefully, it has become the format for exchange as well as for archival storage.

We begin by identifying and evaluating the design criteria first for the underlying raster format, then for tools. The paper concludes by demonstrating the synthesis of a new function (digital halftoning) through application of the atomic tools.

BACKGROUND

The widespread availability of digital raster devices has spawned a large progeny of "raster formats", often with no unifying design principles. It is not uncommon for a format to represent a digital "dump" (on external media), patterned after a device's (or program's) internal data structures. It can be argued that this raster format is finely tuned to the hardware characteristics of its respective device, with subsequent advantages in terms of run-time efficiency.

Our findings do not support this argument. Rather than allowing the specifics and availability of hardware or software to drive our choice of design, we make an *a priori* raster file design, and then argue its advantages. We begin by identifying useful design criteria for both the file format and for the general software system in which it is employed.

In contrast to some proposed formats, our design philosophy has been toward a universal file format which is "moderate" in providing sufficient attributes to model any display device, but without any unnecessary or redundant file attributes: it is minimal. This philosophy encourages the construction of tools which embody raster functions in as abstract a setting as possible. As a direct consequence, tools which deal with only a subset of valid raster files will not exist. This approach is a major departure from many other raster systems.

DESIGN OF THE FILE FORMAT

Raster Specification and Operation

No raster specification is present that might be ambiguously interpreted as a raster operation. Thus, "width" and "height" are essential raster specifiers; raster "orientation" is not, because the raster rotation function exists as a tool, and thus is not (by design) a specification. As a consequence, the raster rotation code belongs to a single tool, which aids in software maintenance. This model frees the user from the dilemma over choice of representation and tool application. In previous systems, a custom tool (e.g., a laser output tool) might accept rasters of only a specific orientation, based on speed considerations. Alternately, that output tool might provide a high-speed implementation of rotation independent of a raster rotation tool. In the first case, the user is left with a question of specification to guarantee operation of the printing tool. In the second, the locus of code which provides raster rotation is not well defined.

Some scene representation languages take the opposite extreme. Functional specification is allowed in the most general sense. Here "tools" don't exist *per se*; their function is present in the interpreter which reads a file. An example is the Xerox Interpress standard [[Spro81]]. Here the general implementation of the file format on a printer implies the existence of supporting code to perform rotation, rendering and all other operations potentially specified by the document. Because this interpreter is monolithic, operations are not free-standing programs. Thus, integration of new software is difficult for a diverse software community.

In our experimental setting, we do not advocate that our raster file allow for "programming" in this sense: we envision situations where one function might be applied to many sets of data, and *vice versa*. Non-radical manipulation of rasters capitalizes heavily on this separation, and we insist on it. In our setting, our well-defined files embody the raster data, and a toolkit of machine-executable files (or UNIX shell scripts) embodies the raster operations.

Pixel Specification

The format provides for the formal specification of a pixel, which allows generic tools (such as crop or rotate) to operate on arbitrary data sets, with independence both from raster dimension and pixel specification. The importance of this should not be underestimated. Historically, formats allow for a maximum of three or four pixel components (often not even within the same file, but as "separates"). Pixel precision is usually taken from a small set, such as one, eight, and twelve bits. Experience has shown that it is impossible to predict *a priori* what or how many attributes comprise a pixel – new models are not to be discouraged. Besides the obvious RGB colour components, traditional data sources often carry multi-

spectral data or Z-depth information. The last few years have brought "alpha" coverage factors and sub-pixel masks to the forefront.

For instance, the Orcatech-based *Palette* system treats pixels as a sixty-four bit quantity, by encoding both foreground and background primary colour information, plus other parameters including masking and transparency, thus modeling artist's use of paint. Our format embraces this experimental system easily. It is worth noting that the system has both a different word length and integer byte order than the original VAX implementation, but this is entirely invisible to the tool creator. Images created by *Palette* may be moved using standard tools (UUCP) to the VAX and rendered on VAX-based graphics engines.

Syntactically, we define pixels as collections of "fields", used to identify the components, in a manner analogous to the record structure type in languages such as Pascal. The pixel attribute is recorded in the file header as a text string. Pixel components consist of an alphabetic identifier and an associated integer which defines the field precision (up to thirty-two bits per component). The identifier is occasionally used externally to specify pixel components to certain software tools. For example, `1mextract` merges and extracts pixel components from multiple input files into an output raster, based on the user-specified **field set.** The precision component is rarely presented to software tools. as the low-level routines allow correct arithmetic operation across files of differing pixel precision (but usually with conforming field names). This is a function unique to our package, and enhances general compositing of files from diverse sources.

Interpretation of Pixel Data

The interpretation of the data fields is at the user's discretion. In many cases, data is taken to span the closed interval `[0..1]`. This interval is closed under multiplication and complementation. The low-level tools provide a data presentation level which returns floating-point values for pixel components on the range `[0..1]`, so the actual field precisions can be kept invisible. This interval is consistent with the design of a number of colour spaces such as RGB, CIELAB and HSB [[Smit78]], in which the three independent colour axes are placed within the interval `[0.0..1.0]`.

Unfortunately, many software tools in existence wrongly (often implicitly) use the interval `[0..1)`. The latter follows directly when software employs bit shifts to map between pixels with differing numbers of significant bits. In that model, a one bit pixel image (to take the worst case, albeit a very common one), [0..1) allows only the intensity values 0.0 and 0.5. When taken to higher significance, the binary value `.1` becomes $.1\bar{0}$. This system never allows "full-on" to be represented.

A useful mapping has two important properties: *reconstruction* and *representation*. Reconstruction means that data can be mapped into any higher precision, and when subsequently mapped back to the original precision, reconstructs the original data exactly. It is not hard to see that bit shifts are lossless operations and therefore have this useful property. Representation means that pixel data of lower precision can be mapped to a system of higher precision, with the pixel values mapping exactly onto identical intensity values. In general, perfect representation is not possible when moving to higher systems, but it can be achieved in many cases, while providing reconstruction universally.

The proper approach regards the interval as being of length 2^n-1. In general, our mapping always provides exact values for intensities `0.0` and `1.0`, so our interval of representation is the *closed* interval `[0..0..1.0]`. Note that binary (one bit) data in our system represent `0.0` and `1.0` exactly. Adoption of this system means replacing bit shifts (multiplies and divides by 2^m) by general multiplying and dividing. This is not a severe speed penalty. In practice, a scaling table can be constructed and a lookup operation used to find the appropriate mapped value. Our method also provides for reconstruction, because a scale up of one bit provides `2n+1` new bins, where `n` existed before, and uniform distribution means that no two values collide.

Exact representation is possible whenever whenever `m` is a factor of `n`. To illustrate this, `4` is a factor of `12`, so we assert that four bit data has an exact representation in a twelve bit system. To prove that 2^4-1 is a factor of $2^{12}-1$, express them as bit streams: '111111111111' can be divided by '1111' giving '000100010001', or 273. Thus, `4095=15*273`, and the representation for white is still exact. More generally, multiplying any value in the four bit system by 273 yields exact representation in the twelve bit system.

Textual Header

Another departure from many "standard" raster file formats is the exclusive use of case-independent, human-readable text within the header. The use of small "binary" headers with magic word values is still common. Yet in raster files, the header typically constitutes less than 1% of the total storage. The advantage we gain is a parser made common to all user software (and thus is part of the low-level raster primitives). Because our header is minimal, this is a simple task. Direct viewing of the attributes of a file means merely viewing the first few lines of it — no special tool is used.

Because both the header and raster data are represented by a byte stream (giving machine independence), we mandate that no "alignment" specifications to the raster be made — the physical

raster immediately follows the textual header. Experimentation with UNIX-based systems indicates that non-alignment to disk boundaries makes almost no difference to software throughput, particularly where the blocking size on disk transfers is large.

The representation of our header data structure in human-readable ASCII text is a trend increasingly common in good software practice. The design of the highly-successful CIF2.0 by Sproull and Lyon [[Hon80]] as a VLSI exchange format mandated use of ASCII to allow electronic mailing of design geometries. The format has gained widespread acceptance outside this realm, as it can be implemented easily on machines of differing character representations and word precisions. Sproull previously designed the Xerox AIS [[Baud77]] raster format (replete with binary header information), and now argues convincingly [[Spro83]] that this trend toward textual representation should be universally adopted, even where the need for exchange is of secondary importance.

Compact Representation

Archival storage of raster images relies on data compaction. Because we desire a universal format requiring no explicit (de)compression steps, our basic format must provide for compression as part of the pixel specification, and implement this operation as part of the basic access routines.

After two attempts at general data compression, we chose a "compaction" operation, which operates on a level beneath pixel specification, and immediately above the level of physical data movement to external media. Our compaction scheme is a general run length coder, which replaces identical runs of n bytes with `n+1` bytes of code, representing the original run, plus a count in the range `[1..256]`. We choose the term "compaction" over "compression", as the operation may take place without regard to pixel boundaries. Early experiments indicate this may have value where images containing data that is cyclic across a scan-line (halftoned images, stipple patterns) are to be encoded. The compression size can then be set to span a collection of adjacent pixels.

General and Special Cases

The raster proper is encoded in a manner which maximizes speed of raster (un)packing by aligning pixel groups onto regular boundaries. Although the specifics are detailed, this underlying code guarantees packing efficiencies of more than 84% for pixel sizes up to 12 bits, approaching 100% for arbitrarily large pixels. This design choice minimizes overhead in the data extraction loop, as the shift and mask values are constant over the data set.

The criteria set forth above allow "special cases" to fall out directly from the more general specification, without special caveats being coded in. This is intentional. For instance, the external representation of an `640x480` size raster of twenty-four bit RGB pixel values is quite simple: a textual header, followed by `640*480*3` bytes of data, arranged in R,G,B order, by scan-line, without any padding. Although we don't advocate that tools write rasters independently of the low-level software which defines the header specification, it does indicate the simplicity and generality of our approach. For instance, a videotex station could dump out a hard-coded header string, followed by a byte dump of its screen contents thus creating a well-formed "cannonical" raster format file.

TOOL DESIGN

General Philosophy

Brooks' findings [[Broo75]] show that as a rule a long-lived systems consist of software which outlives the intention of its original use. Because we cannot anticipate the user's ultimate needs or goals with the raster tools, we should choose to craft each tool into an atomic, composable function with no implicit assumptions of the user's intentions. From this "metaobjective", a number of practical considerations immediately become clear.

Consistency of Design

Overall design consistency leads to tremendous ease of use for both implementor and user. In particular, the time spent in learning the tools used for simple operations becomes proportional to the user's objectives; what little "start-up overhead" exists is common to all tools, and need not be relearned. Similarly, the tool designer can fashion a new tool based on the existing package, thus implicitly inheriting uniformity of the user interface (such as commonly used command line switches) and operation.

As an example of consistency, all software tools dealing with the concept of an axis-aligned rectangle specify this entity in terms of origin and size, not as diagonally opposed corners. In contrast, corner-based specifications leave the ambiguity of semi-open intervals for the user to resolve. For instance, the corner specification model might describe a `512x512` display as spanning the region `(0,0)` to `(511,511)`, whereas our model describes the display as a window of size `(512,512)` with origin location `(0,0)`. Thus, we remove the burden of potential "off by one" errors; in fact the casual user will probably be unaware that any ambiguity could exist. This "correctness by design" is a very powerful concept in implicitly steering the user along a correct path of conceptualization.

Minimal Atomic Set

The tools are atomic, composable functions which deal with raster data in the most abstract way conceivable for each respective function. They strongly resemble Guibas's concept of a bit map calculus [Guib82] with the accompanying language MUMBLE. Our implementation provides for pixels of arbitrary precision, as do his; our "language" consists of UNIX commands in which tools play the part of keywords to perform manipulations on the data. A compiler for a large subset of MUMBLE using the toolkit as "machine code" would be a straightforward exercise.

Just as computer languages advocate a small number of composable keyword constructs, we encourage the user to synthesize function from tools already within the kit. When this fails, he should seek the most general tool necessary to extend the coverage of the tool set to contain this specific operation. Besides allowing for a new function with the least amount of new software, a minimal addition to the toolkit can be very revealing to the deep structure of the problem.

THE TOOLS

Although space does not permit a description of each tool, we may summarize the operation of the toolkit. It is useful to characterize classes of tools by common operation. These form our taxonomy.

Storage Considerations

Because tools are composable, they operate on data presented serially. However, some tools require internal raster storage to perform their intended operation. We classify these as "level 0" (constant pixel storage needed), "level 1" (constant scan line storage needed), and "level 2" (arbitrary storage). In each case, these are worst-case raster storage requirements. A generous number of tools belong to classes 0 and 1. Sequences of operators may therefore manipulate images on secondary storage whose size exceeds system main memory.

Input/Output Characterization

Because tools are operators, we may classify them as "unary", "binary" or "tertiary" tools, based on the number of simultaneous inputs used. Two additional classes: "source" and "drain" represent tools which interface between the toolkit universe and some other means of representation. These include display output tools, text input tools and pattern generators. With the exception of drain tools (these typically render images), a tool will have one "standard" output in the form of a universal image file. This class characterization is formally specified in the source code of each software tool, thereby activating library routines common to all tools within this class. Such code governs the number of expected input files and enables command line switches generic to that class.

Other Characterizations

A final characterization is whether a tool preserves pixel integrity. Most tools do, and this is important when a tool is used to manipulate non-intensity pixel fields, particularly when the tool is used in settings far removed from traditional imaging applications. Cropping can be performed on data which contains Z (depth) information, but a low-pass filtering of such data is not intuitive because the latter does not preserve pixel integrity. At present, few tools which violate pixel atomicity exist. Pixel-preserving functions fit in well with our design philosophy of generic tools.

TOOLS BY EXAMPLE – HALFTONING

The following examples demonstrate a series of experiments used to perform digital halftoning (the creation of bi-level images) from high resolution sources. The presentation is an idealized "lab session", but it is also a recapitulation of the historical development of the tools.

Experiment #1 – Plate #1

We begin by halftoning through simple thresholding. We envision thresholding as a binary tool which does a test for magnitude of its two inputs. The source tool `imconst` is used to provide a reference level for the secondary input. Generally, thresholding can be modeled as a subtract operation, with a subsequent test to map `x>0` values into white, else black. This last function already exists as `imtomask`. We thus perform the test `a>b` as `(a-b)>0` stepwise on the image file containing a "milkdrop":

```
imconst -d milkdrop.im -v 128 |
  imsubtract milkdrop | imtomask >out
```

Experiment #2 – Plate #2

Thresholding to a uniform, constant value produces poor (as expected) images, so we write a program `imhalftone` to do ordered dithering, comparing input pixels against a cyclic, periodic set of dynamic threshold points to vary the thresholding [Jarv76]. The program is not clean: the `4x4` matrix of threshold weights is hard coded, and the software must permute the array internally to conform to the arbitrary widths and heights of the input file.

```
imhalftone milkdrop.im >out
```

Experiment #3

The halftone results look good, but we need avenues of further exploration. The permuted internal weight table replication code is in fact a "tile" operation first conceived for use with much larger images. We write `imtile`. As a bonus, the tiler is fast, and includes offset switches to be fully general. These correspond to phase shifts in the halftoning dot, a desirable feature in colour digital halftoning, in which

halftone screens for successive colour separations are staggered with respect to previous screens. The threshold table is now recoded as the file `kernel.im`.

```
imtile kernel.im -w 128 -h 128 |
  imsubtract milkdrop | imtomask >out
```

Experiment #4

The 4x4 kernel, now represented as `kernel.im` in Experiment #3 was cumbersome to make. We had also planned software which would to do a "text dump" of raster files; here we wish to do the opposite: convert the "dump" into a raster representation. Realizing that the scope of this software is more than merely one of diagnostic service, we write both `imtabin` and `imtabout`. The hard coded ASCII constants of `imhalftone` have now found a niche. One can envision a standard tool sequence (e.g. a UNIX shell script) to halftone arbitrary images against a textually encoded table of weights.

```
imtabin -w 4 -h 4 -p n8 >kernel.im
240  176   80  208
 96   16   48  128
160   32    0   64
192  112  144  224
^D
```

Experiment #5 – Plate #3

The typing of constants in Experiment #4 suggests a mechanized means to generate random numbers, and we are curious to see the appearance of such output. The creation of an array of random numbers (not specific to halftoning) is all that is needed: the other code is already in place. We rewrite of copy of `imconst` which substitutes random values for constants. The `-default` switch in our example borrows the dimensions and pixel specification of `milkdrop.im`, so that `imrandom` can produce conforming output. The output shows superimposed high-frequency noise [[Robe62]], resembling the grain in film emulsions "pushed" too far during development.

```
imrandom -d milkdrop |
  imsubtract milkdrop | imtomask >out
```

Experiment #6 – Plate #4

The results inspire the use of a thresholds with Gaussian distribution. We recall that the "coin tossing" method [[Kalb79]] generates such sets by averaging small sequences of evenly distributed numbers. This requires binary operators other than subtract (such as average and sum), so we extend the scope of `imsubtract`. The tool is renamed `imaop` because it now allows for arbitrary arithmetic operations from two input sources. We also rediscover that rerunning `imrandom` generates identical values, so we employ `imcrop` to give us a set of different random numbers.

```
imrandom -d milkdrop -h 512 >master
imcrop master -y   0 -h 128 >m1
imcrop master -y 128 -h 128 >m2
imcrop master -y 256 -h 128 >m3
imcrop master -y 384 -h 128 >m4
imaop m1 m2 -op aver >t1
imaop m3 m4 -op aver >t2
imaop t1 t2 -op aver >gauss
imaop milkdrop gauss -op sub | imtomask >out
```

Experiment #7

The brevity of most command lines is pleasing, but `imtomask` is ever-present. Because it is a unary operator immediately following `imaop` in each case, we extend `imaop` to include a `thresh` function. The code integration is trivial: three additional lines and a new switch statement label. As a bonus, the thresholding works "automatically" for colour files – a feature unanticipated at the outset. Thus, `imhalftone` has now been made obsolete.

```
imaop milkdrop gauss -op thresh >out
```

Conclusions

The evolution of the imaging software demonstrates a few important principles. For one, generality of function allowed a means to verify hypotheses, without any programs having to be written. As a clearer understanding of the desired goal emerged, specific tools were crafted. For instance, we created random numbers with Gaussian distribution by a simple synthesis of operation, thus allowing the user a glimpse into their properties. Should this become a desirable feature to support in general, `-gauss` or `-seed` switches may be added to `imrandom`, but for current applications this is unnecessary.

These examples show that when a new operation is sought, it can often be melded into the function of a tool in existence, thus widening the scope of operation for the original tool. This creates a tremendous synergy of function. By studying why more than one path toward a goal exists, we can both pare down what constitutes a minimal set, and simultaneously get new insights into the deep structure of the problem.

Acknowledgements

The complete raster toolkit has been distributed on a limited basis to a number of installations. A more formal release is in preparation, pending the completion of a comprehensive technical report.

The authors wish to thank the many members of the Computer Graphics Laboratory who commented on this research, especially Michael W. Herman with whom many discussions were held during the preliminary stages of the design. The research reported here was supported by the Natural Sciences and Engineering Research Council of Canada under a variety of grants. The first author was partially supported by a University of Waterloo bursary.

REFERENCES

[Baud77] Baudelaire, P., Israel, J., Sproull, R. "Array of Intensity Samples – AIS" Xerox PARC Internal Report, February 1977 (Rev. by K. Knox, 1980)

[Broo75] Brooks, F. P. Jr. *The Mythical Man-Month – Essays on Software Engineering* Addison-Wesley, Reading, MA (1975) pp. 120.

[Floy75] Floyd, R. W., Steinberg, L. "An Adaptive Algorithm for Spatial Gray Scale" *Society Inf. Displays* Int. Symp. Digest of Technical Papers (1975) pp. 36.

[Guib82] Guibas, L., Stolfi, J. A Language for Bitmap Manipulation" *ACM Transactions on Graphics* **1(3)** July 1982, pp. 191-214.

[Higg85] Higgins, T. M. "A Cel-Based Model for Paint Systems" Master's Thesis, University of Waterloo, Waterloo Ontario, May, 1986

[Hon80] Hon, R. W., Séquin, C. H. "A Guide to LSI Implementation" Xerox PARC Bluebook **SSL-79-7** (2nd edition January 1980).

[Jarv76] Jarvis, J. F., Judice, N., Ninke, W. H. "A Survey of Techniques for the Display of Continuous Tone Pictures on Bilevel Displays" *Computer Graphics and Image Processing* **5(1)** March 1976, pp. 13-40.

[Kalb79] Kalbfleish, J. G. *Probability and Statistical Inference* (Vol I), Springer-Verlag 1979, **Sec 6.7** pp. 234-239.

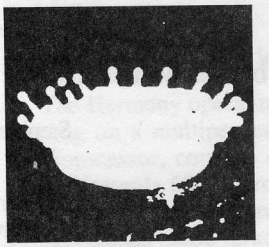
[Robe62] Roberts, L. G. "Picture Coding using Pseudo-Random Noise" *IRE Trans. Info. Theory* **IT-8** (February 1962) pp. 145.

[Smit78] Smith, A. R. "Color Gamut Transform Pairs" *ACM Computer Graphics* (SIGGRAPH '78) **12(3)**, August 1978, pp. 12-19.

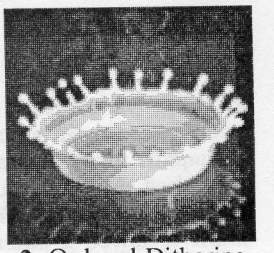
[Spro81] Sproull, R. F., Lampson, R., Warnock, J., Reid, B. "Interpress: A Standard for Communicating and Storing Print Graphics" Xerox PARC Private Document **ISL-81-1** (Subsequently released and made available by Xerox Corp).

[Spro83] Sproull, R. F. *Private Communication*, Xerox PARC, May, 1983.

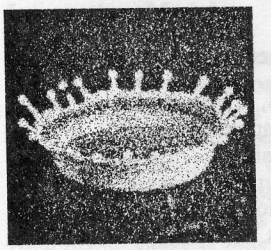
PLATES



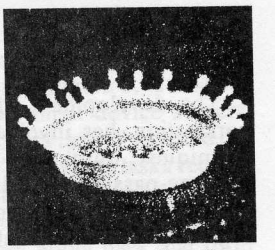
1. Simple Threshold



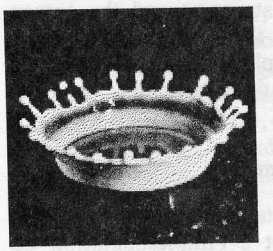
2. Ordered Dithering



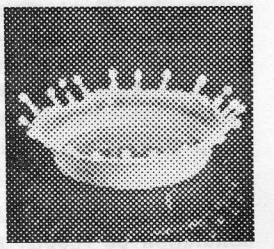
3. Linear Random



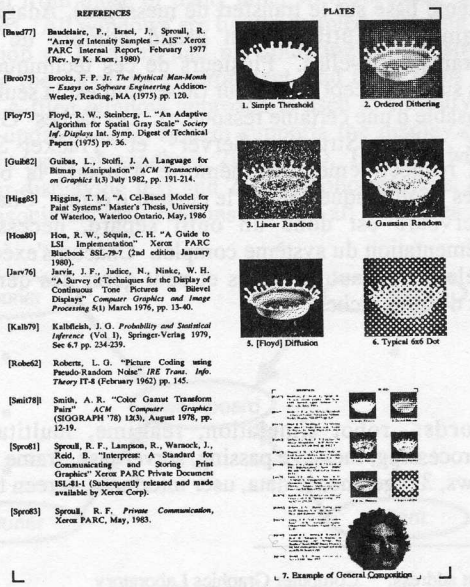
4. Gaussian Random



5. [Floy75] Diffusion



6. Typical 6x6 Dot



7. Example of General Composition