Please retain
for future
copying.

The IM Raster Toolkit
Design, Implementation and Use

Alan W. Paeth

CS-86-65

December, 1986

UNIVERSITY OF WATERLOO
COMPUTER SCIENCE DEPARTMENT

The IM Raster Toolkit
Design, Implementation and Use

Alan W. Paeth

Computer Graphics Laboratory, Department of Computer Science
University of Waterloo Waterloo, Ontario, N2L 3G1 Canada.
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 format 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.

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

The research reported here was supported by the Natural Sciences and Engineering Research Council of Canada under a variety of grants, a University of Waterloo bursary and an Institute for Computer Research scholarship.

Contents

List of Tables

List of Figures

The IM Raster Toolkit
Design, Implementation and Use

Alan W. Paeth

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

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 technical report discusses the design, implementation and use of a comprehensive raster manipulation system, based on a 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.

The first section (Design Issues) is a familiarization of the work and its underlying motivation while placing it within the historical context of raster-based imaging systems. The section also illustrates the practical use of the toolkit while omitting fine detail. This is done through the presentation of an annotated "lab session" in which digital halftoning experiments are conducted.

The second section (Implementation) provides a in-depth discussion of the file format and the body of software used to manipulate it. Its intended audience are implementors and maintainers of the system's software. The section is annotated with functional examples (C language fragments) which depict the creation of both stand-alone and toolkit-based programs.

The third and final section (Tools Description) provides exhaustive documentation, listed alphabetically by tool name, in a format resembling Unix "man page" on-line documentation. The section serves as the complete reference for all software tools included in the Image Tools tape, distributed in parallel with this technical report. The present version is available in 9-track 1600 bpi media. Contact:

Computer Graphics Laboratory
c/o Dept. of Computer Science
University of Waterloo
Waterloo, Ontario, N2L 3G1
CANADA (Tel: 1-519-888-4534)

Design Issues

1.1. 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 frame buffer hardware, including those built by Adage/Ikonas, Raster Technologies, Apple/Macintosh, Atari/ST, VAX/GPXII, plus raster output engines, such as the Versatec and Imagen hardcopy printers. In their 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. Run time initialization code is used to test the specifics of the machine 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* [Higg86] color painting system. Here the user may disregard the different pixel precisions, (lack of) color, or machine word-lengths, all of which differ for each hardware system. The operation of painting a bitmap image has conceptual meaning and the generic tools support it directly. Because the file format has been designed carefully, it has become the format for exchange as well as for archival storage.

1.2. Background

The widespread availability of digital raster devices has spawned a large progeny of "raster formats". We may well expect this trend to continue. Unfortunately, without considering what comprises a general "raster image", there are often no unifying design principles in designing general raster-based systems. 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 fine 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 a minimal specification. This philosophy encourages the construction of tools which

embody raster functions in as abstract a setting as possible. As a direct consequence, the low-level support software can then operate on any raster within this format – tools which deal with only a subset of valid raster files will not exist. This leads to two major departures from many other file formats.

1.3. Comparison with Other Formats

Previous formats can be characterized by their absence or overabundance of data specification. In either case, this leads to weaknesses when tools are subsequently designed and orchestrated together to form a working system. We give examples of each.

1.3.1. Xerox AIS Raster File Format

The Xerox AIS (Array of Intensity Samples) format [Baud77] provides a comprehensive binary header to specify raster attributes, followed by the raster data. No mechanism for compression or encoding is present. Instead, provision is made to "performance" tune the file for high-speed disk operations. Specifically, periodic insertion of empty space after a sequence of n scan lines causes the data to realign modulo some base after every n lines, where n is specified in the header. This speeds the inner loop of some critical algorithms, allowing a disk to produce data faster than a free-running laser printer can consume it. Unfortunately, software which supports the full generality of this format can become intricate, and in fact various existing display tools (for output on bi-level CRTs) cannot accept such files.

The pixel specification of the file allows precisions of one, two, four, eight and sixteen bits of precision, with an optional specification to group multiple separations in a pixel. In practice, virtually all data representations are either eight bit color separations or one bit monochromatic rasters, and this is reflected in many tools' lack of generality. Full color separations are always encoded in different files; this is another concession to color xerography, allowing consecutive color toner passes to be paired with an accompanying color separation file.

In comparison to our work, the format has some weaknesses. The disk sector performance tuning appears unnecessary under UNIX (see Table 1). Arbitrary pixel precision cannot be captured and even those precisions allowed are not fully supported. Finally, no "clean" mechanism is provided for dealing with color separations within one file. In fact, a common convention is to embed file attributes within the file name: (e.g., "JupiterFlyby-RED.ais"), but no uniform convention is mandated. An even more serious liability with the AIS format exists in the ambiguity of specification present in the header. This is the subject of Section 1.4.1.

1.3.2. Lucasfilm Raster Format

Other recent formats are similarly driven by specific user needs. For instance, the approach used by Lucasfilm [Port83] optionally breaks a picture into $N \times M$ tiles, so that windowing operations into this aggregate data structure can attempt to minimize disk transfer operations. In this setting, the assumption is that only a small portion of a potentially large image is active at one time, as in the case of generating stage "flats" for use in films. Although this approach cannot be faulted *per se*, it can become cumbersome in a general, experimental setting. Here, scan line based operations such as digital filtering and image convolution literally cut

"against the grain", as any one scan line potentially spans a number of adjacent tiles. Because these latter algorithms are so common, a general implementation of this format suggests a body of underlying code to reassemble scan lines properly. But this merely suggests that the underlying file representation simply be a single raster. The Lucasfilm format allows channels of only eight or twelve bits, with bit flags specifying channels taken strictly from the set { *red, green, blue, alpha* }. This hints at departing from the mere encoding of color values within a pixel, but does not capture the full generality of this concept.

1.3.3. Exchange Formats

A number of exchange formats exist, which typically consist of (optionally) a header and trailer part, together with the raw "dump" of the raster data, typically packed into eight bit bytes or short (sixteen bit) or long (thirty-two bit) words, with end-of-line or end-of-block data occasionally present. Almost without exception, a pixel consists of one eight or twelve bit value, giving intensity for one visual component, such as monochromatic intensity, green intensity or infra-red intensity. The USC image library and the ERDOS Landsat image data are examples of such exchange formats. As a rule, these formats encode "color separations" as physically distinct files, with the composite images formed from a consecutive set of magnetic tape files. Here the separation entity (also called the "band" or "channel") is implicit in the tape file order, but in some cases a single character identifier or a flag bit indicates the proper band. In a small number of cases, the pixels are "composite" with all banks aggregated within one pixel. Fortunately, the low-level nature of these formats allows them to be easily converted into our more general format, with no loss of information. In fact, implicit data such as tape file ordering can be recovered and subsequently become a useful attribute.

1.3.4. Exotic and Experimental Formats

Exotic formats typically encode data where the sample points are not arranged as a square or rectangular grid. For instance, US Weather Service raster grids are in Stereographic projection, allowing any ground circle (locus of points in a cylinder normal to and intersecting the world sphere) to project as a circle on the grid – her a grid rectangle would conform to a strange pin-cushion in the spherical domain Landsat data can also be acquired for survey work in a UTM (Universal Transvers Mercator) projection; this projection is not even conformal (angle preserving), in contrast to standard Mercator and Stereographic projections. Here the sample gri can be viewed as a distorted rectangular mesh. Similarly, raster data which encode a non-rectangular tiling of an image (equilateral triangles and regular hexagons ar well-known examples) have been suggested, and are worthy of furthe experimentation. In each case, a general specification should be flexible enough t store the data without any undue gyrations, but leave the specifics of dat interpretation to those tools designed specifically for this setting.

1.3.5. Related and Contemporary Work

The adoption of Unix pipes to realize composable imaging tools was first proposed by Stevens and Hunt in 1981 [Stev82]. Their extensive analysis of throughput timings (on a PDP11-70) show a consistent gain of 60% by virtue of temporary file elimination. Their work provides in-depth discussion of the underlying Unix pipe mechanism (buffer pools and asynchrony) used to support their findings.

The Unix/VAX based HIPS system [Land83] actualizes this proposal and provides an extensible textual header. Their header allows the definition of arbitrary pixels, which permits experimentation in image data structures, such as oct-trees or pyramids [Tani80]. This shifts the burden of pixel interpretation to the tools, thus restricting certain file sub-types to specific tools and giving rise to specialized conversion software (e.g., "btof" to convert byte to floating-point data). The work is exemplary in embracing traditional image processing techniques used in image filtering, transforming, coding, compression and edge-finding.

On-going work by Havens, *et al* at the University of British Columbia provides a clear separation between format specification and tool operation. Their file format is built upon an independent "bitio" mechanism [Have82] which allows for I/O across Unix pipes in blocks of arbitrary bit length. Their textual header specifies this pixel length, together with raster dimension and photometry information. While their work recognizes that two-dimensional sampled data may encode properties other than intensity (such as simultaneous R,G,B values or depth information), their file format nontheless lacks the concept of pixel fields. Thus, the format is used to manipulate single-valued (e.g., monochromatic) data in the traditional image-processing setting.

1.4. Design of the File Format

1.4.1. 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 but 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 application of tool. 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.

At first encounter, a minimal specification seems unsuitable for use in practical applications. For instance, allowing the inclusion of a color map specification would let the user deal with minutiae such as device gamma correction. But in fact, its inclusion would only compound the problem, because the specification is now exploited to mask out device dependency, with no explicit reference to this dependency within the header. Our approach advocates the addition of a *-gamma* switch to any display tool used to render image files. The latter might well make

the user's perceived need for color maps vanish. Further, files may now be reimaged on devices with different ranges of gammas (e.g., CRT film recorders) with very little effort, whereas this would be very difficult given header-based color tables precorrected for an expected value of gamma. It should be noted that the *-gamma* switch on the display tool might well employ the hardware color maps of its display device to provide the desired effect without need to recode the data sent to the display, but the user need not be aware of this. Here a clean line of separation exists between the functions of color mapping and gamma correction. Each is supported by an appropriate piece of software, and the cleaner conceptual model benefits the user enormously.

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. Examples are the Xerox Interpress standard [Spro81] and its direct offspring, PostScript [Adob85]. 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.

1.4.2. Universal Format for Exchange

The format must provide for a universal means of data (i.e., image) exchange. For our purposes, this exchange will most often be in the form of either twenty-four bit color or one bit monochromatic images, and these cases should "fall out" of the general specification, with no explicit use of special rules. Further, "universal" must embrace a growing number of color and monochromatic raster devices, each with its own specific pixel formats and raster dimensions. Exchange might well be between different machines and different raster types; it must accordingly be described as a data structure which does not rely on programming or algorithmic constructs present in some specific programming language or hardware device. In fact, because the original setting (the Computer Graphics Laboratory at Waterloo) has a large number of devices, it is a basic criterion that the format run on all raster devices, thus making movement of images from one display to the next a very simple task.

To be a practical format for exchange and storage, the format must remain general while still guaranteeing a high degree of storage efficiency, particularly where files can be extremely large. Our specification provides a worst-case packing efficiency of 71%, with a typical efficiency of >81%. In eight common pixel sizes between one and thirty-two bits, our packing ratio reaches 100% (Figure 1). Provision for general run-length compression is also made.

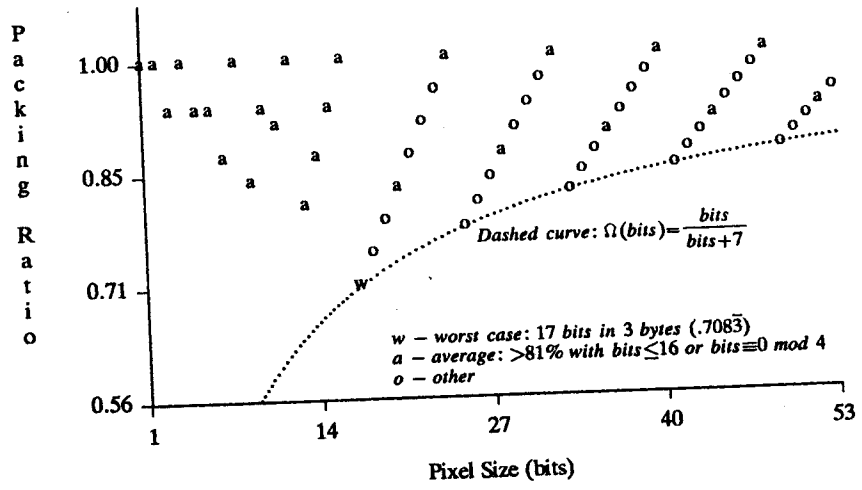


Figure 1 – Storage Efficiency

1.4.3. 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 allowed 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 color components, traditional data sources often carry multi-spectral data or Z-depth information. The last few years have brought "alpha" coverage factors [Port84] and sub-pixel masks [Fium83] to the forefront.

For instance, the Orcatech-based *Palette* system treats pixels as an eighty-four bit quantity, by encoding both primary color information, plus other parameters including coverage and transparency, thus modeling artist's use of paint. Our format embraces this experimental system easily. It is worth noting that the Orcatech 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, *imextract* merges and extracts pixel components from multiple input files into an output raster, based on the user-specified field set. The precision of a 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.

1.4.4. 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 complementation (with $\bar{x}=1-x$) and multiplication. 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 color spaces such as RGB, CIELAB and HSB [Smit78], in which the three independent color axes are placed within the interval [0.0..1.0].

Unfortunately, many software tools in existence implicitly use the interval [0..1). This is simply incorrect. The latter follows directly when software employs bit shifting 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 .10$\bar{0}$. 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 shifting is a lossless operation and therefore has 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. Here the shift approach clearly breaks down: white (all bits on) in any system can never represent full white in a system of higher significance. In general, perfect representation is not possible when moving to higher systems, but it can be achieved in many cases, while providing reconstruction universally.

Our approach notices that the failing with the bit shift approach is in representing the n bit values $0..2^n-1$ on an interval of length 2^n, which is implicit when using bit shifts to multiply pixel values by powers of two. The proper approach regards the interval as being length 2^n-1 long. As an example, with $n=8$, black and white in our system are 0/255 and 255/255 respectively, and not 0/256 and 255/256. 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 multiplication and division. 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. Representation is more difficult to achieve.

When scaling up to a system of exactly twice the number of bits, it is guaranteed. Here the higher system $2^{2n}-1$ can be factored as $(2^n-1)(2^n+1)$. The left-hand factor is the lower system, and the right-hand factor is simply an integral scale value. For instance, $n=4$ bit data represents intensity values spaced by 1/15, and when scaling to $n=8$ bit data, $2^n+1=17$, so we multiply by 17 to scale up. Here white (15) becomes white (255) because $255=15\times17$.

In general, exact representation is possible whenever an integer scale factor exists to map to the higher system. This means that white in the lower system must factor into white in the higher system. Fortunately, this is not only the case when the number of bits in the higher system is a doubling of the original bit length, but when it is any multiple of it. To illustrate this graphically, 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, $4095=15\times273$, 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.

1.4.5. Textual Header

Another departure from many "standard" raster file formats is the exclusive use of case-independent, human-readable text within the header. The urge to write fractionally smaller "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 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 (a choice which gives wide machine independence), we mandate that no "alignment" specifications to the raster must be made – the physical raster follows directly after the end of the textual header. Experimentation with UNIX-based systems indicate that non-alignment to disk boundaries makes almost no difference to software throughput, particularly where the blocking sizes on transfers to and from disk are large. This is shown in Table 1.

BUF LEN	COUNT	RAW I/O		BUFFERED I/O	
		aligned	**non-aligned**	aligned	non-aligned
128	8000	12.5s	**12.7s**	15.5s	15.3s
512	2000	4.3s	**4.3s**	15.4s	15.7s
2048	500	2.0s	**2.2s**	15.3s	15.3s

Table 1 – Low-level Disk Performance

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.

1.4.6. 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 must 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 replication count in the range [0..255]. We choose the term "compaction" over "compression", as the operation may take place without regard to pixel boundaries. This has value where encoding images containing cyclic data (such as stipple patterns), or when encoding pixels of sub-byte precision, as in binary images. The compression size can be then set to span a collection of adjacent pixels.

Returning to a previous example, the *Palette* system first adopted the package because of limitations of external storage, further compounded by its unusually long, exploratory choice of pixel specification. Because the tool support code allows for compaction of arbitrary pixels, the file I/O mechanism was completed in one afternoon. As an added benefit, possibilities such as halftoned laser hardcopy became immediately available.

1.4.7. Simple and Special Cases

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 640×480 size raster of twenty-four bit RGB pixel values is quite simple: a textual header, followed by $640\times480\times3$ bytes of data, arranged in R,G,B order, by scan line. No special padding is required. Even in the case of compressed data, our scan line format matches the compression scheme used both for binary images on the Apple/Macintosh and for RGB images on the Raster Technologies frame buffer, and these specific tools thus reap performance advantages.

In a number of cases a "raw" raster file from an external source can be made to conform to the file format merely with the textual prefacing of some "boiler-plate" which serves as a header. Although we don't advocate that tools write rasters independent of the low-level software which defines the header specification, it does

indicate the simplicity and generality of our approach. It allows for the generation of valid image files by software that otherwise cannot support the entire low-level image library. For instance, a small microprocessor running within 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.

1.4.8. Summary

We may briefly summarize our design criteria:

- No forward referencing allowed – one pass sequential reading.
- Attribute header in ASCII, with specification through keyword and value.
- External representation a sequential byte stream – no "skips" allowed.
- Support of compression allowed to achieve good off-line storage.
- Specification of width, height is present.
- Specification of both pixel fields, plus field precisions is present.
- Specification information is minimal.

1.5. Design of the Tools

1.5.1. General Philosophy

It is not sufficient that code be well written, useful, and accepted by a user community. Brooks' findings [Broo75] show that as a rule 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.

1.5.2. 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 use command line switches) and operation.

The domain of this last constraint is larger than the mere specification of the file format. It argues strongly for uniformity of design, convention and paradigm throughout. Taking examples of each uniformity: the file specification is uniform by design, the command line switches are uniform by convention, and the use of UNIX pipes to concatenate raster operators in a manner analogous to their traditional use (the concatenation of text operators) is uniform by paradigm. Taken together, this uniformity greatly eases the mental load on all users. In many cases the use and invocation of a software tool can be inferred with high probability of success by the casual user.

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. This choice is consistent in that it encodes a notion of dimension

directly (width and height), as do all raster files. In contrast, corner-based specifications leave the ambiguity of semi-open intervals for the user to resolve. For instance, the corner specification model describes a 512×512 display as spanning from (0,0) to (511,511), whereas our model describes the display as a window of size (512,512) with the origin offset to location (0,0). Thus, we remove the burden of "off by one" errors which might potentially plague a user; in fact the casual user will probably be unaware that the potential for ambiguity exists. This "correctness by design" is a very powerful concept in implicitly steering the user along a correct path of conceptualization.

1.5.3. 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 bitmap 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. This is seen later in Section 1.8.

1.6. Tool Characterization

We summarize the operation of tools within the toolkit. A description on a tool-by-tool basis appears in the next chapter. It is useful to characterize classes of tools by common operation. These form our taxonomy.

1.6.1. 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" (constant raster, or arbitrary scan line storage needed). 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.

1.6.2. Input/Output Characterization

Because tools are operators, we may classify them as "unary", "binary", "triple" or "multiple", 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.

1.6.3. 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 meaningless because the latter does not preserve pixel integrity. At the moment, few tools which violate pixel atomicity exist. The predominance of pixel-preserving functions fits in well with our design philosophy of generic tools.

1.7. Standard Tool Library

Whereas the underlying file format is rigid, our system expands as new tools are created. The present system relies on a tool building library to provide a uniform tool interface (including helpful diagnostics) to the user. The majority of users of the package who will write code will interface with the access routines at this level. Users are encouraged to redesign or extend the tool library and its user interface, so long as it does not make any changes to the underlying file format of images. The conventions and syntax of our tool library is discussed below.

1.7.1. General Conventions

The names and conventions of each software tool are highly systematized. Much of this is a direct consequence of the common body of low-level routines, but in some cases this consistency of use is enforced by the creator of the tool. For instance, the switch names *-width*, *-height*, *-xoffset* and *-yoffset* (and any proper initial substring thereof) are present in a great number of programs. In the most general setting, their meaning is to present the output data from their tool using the offset and dimensions given. In cases where the output and input size conform, and some auxiliary data set is provided, they often provide further information on how the primary and secondary input sources combine. Although the second half of this meta-rule seems vague at first, it in fact indicates that the user can very often infer the meaning of the offset switches and provide accompanying values, without having to rely on a prodigious memory. This is consistent with the function specification paradigm of the VLSI design editor Caesar [[Oust81]]. In that system, virtually all raster operations require a rectangle as an operand. In simple cases, the rectangle specifies layout regions to be moved or copied. In more abstract settings, the cursor

grid pitch and offset can be defined based on the rectangle's size and origin, respectively.

As an example of the use of offset and dimension switches,

```
imconst -w 10 -h 20 >out.im
```

generates a 10×20 raster of constant full intensity with output to the raster image file "out.im". Here offset data is not allowed; offsetting data is a function applied when moving rasters about, not an attribute of any single raster. Similarly,

```
imikw -x 10 -y 20 <in.im
```

outputs the full contents of the raster image file "in.im" to the Ikonas frame buffer, with the upper left-hand corner of the raster image offset by (10,20) from the similar origin of the display screen. (Note: this choice of origin is common to raster manipulation software and is shared by matrix (row column) notation. See also Section 2.2.3). Here width and height attributes are meaningless. A third command,

```
imcrop in.im -w 30 -h 20 -x 4 -y 4 >out.im
```

crops (extracts pixel by pixel) a rectangle with origin at (4,4) and of size 20×30 and sends the output to the standard output, in this case the raster image file "out.im".

Examples of the other switches (in a spirit similar to the Caesar example, above), are:

```
imtile <in.im -x 4 -y 2 -w 512 -h 512 >out.im
```

This program replicates an input raster image file (without gaps) to form a "quilt" over the output raster image file. The input dimensions are known from the input header, the output dimensions are specified by the *-width* and *-height* switches; *-xoffset* and *-yoffset*, (though perhaps not obvious at first) are the "phase" offset defining the location of input pixel (0,0) in the output file. Note that the novice user might not even be aware that such offset switches exist (his conception of tiling does not include this generality), but upon realizing the importance of offsetting a tile (it commonly occurs in digital halftoning) he might correctly infer that *imtile* accepts *-xoffset* and *-yoffset* switches. The conventions here are "reasonable" interpretations of what a "rectangle" means to various software systems.

In other cases, ambiguity of meaning in program operation can be removed by choosing the alternative most in keeping with uniformity of overall system design. For instance, a program to run "movies" on a frame buffer by packing multiple images (for example, sixteen 256×256 images of eight bit pixels into one 512×512×32 bit frame) must make a choice about the nesting of loops in enumerating the frames. The consistent choice here is:

```
by Height do:
    by Width do:
        by Pixel do:
            nextframe(parameters)
```

so that the loop order corresponds to the conventional loop order used in creating a single image; the innermost loop selects between images within one frame buffer pixel. Thus, the images begin with 1 in the upper left-hand corner (together with its

companions, 2, 3, and 4), and end with 13, 14, 15, 16 in the lower right-hand corner, which corresponds to an intuitive layout (because the design choice of loop order in the file format was also intuitive).

1.7.2. Switch Conventions

A number of switches are present in many programs, often with some generic meaning. These are summarized in Table 2.

SWITCH	MEANING
-compress {int}	output file compression value
-default {file}	default specifications taken from "file.im"
-height {int}	height specification in output header
-pixel AiBjCk...	pixel specification in output header
-usage	general help/use message
-value x:y:z	specifies [0..2^n−1] field values x,y,z...
-width {int}	width specification in output header
-usage	generates list of all switches and defaults
-xoffset {int}	x location of the output data
-yoffset {int}	y location of the output data

Table 2 – Common Switch Names

The *-usage* switch is accepted by *all* software tools, and is particularly helpful as a quick reminder of program operation. Typing a tool name alone typically results in its waiting to read an image file from the standard input (the keyboard), with dimensions and other switch values defaulting – a rather useless operation. Many programs require height, width and pixel switches. Tools requiring all three (e.g., "source" tools) may use the *-default* switch, which reads merely the header specification of the file, specified by the argument following *-default*, and substitutes the *-width*, *-height* and *-pixel* default values accordingly. These may in turn be overridden by subsequent explicit command line switches. The *-default* switch is useful in defining secondary input or output which conforms in size and pixel specification to some primary input. The *-compress* switch specifies the compression of the output data; it defaults to the compression setting of the input file. The *-value* switch provides exactly one pixel value of input in textual fashion; it defaults to full value for each component present. For instance, the pixel specification "r8g8b8" defaults to a 255:255:255 setting.

Other switches not present in the table typically appear in display output tools. For instance, *imikw* has *-res* and *-zoom* switches for high resolution mode and zoom mode, respectively. As a rule, tools use only those switches that are present in this table. This not only aids use, but is indicative of the high level of abstraction achieved by the tools.

1.7.3. File Name Conventions

The system makes the following assumptions about input and output files. First, all output is without exception to the standard output, with error diagnostics directed to the standard error output. Almost without exception, the format of the output data is an image file. Where input is required, it is generally taken from the standard input, unless a file name appears immediately after the tool name, in which case this name specifies the data set. On programs requiring two inputs, either two file names are given or the standard input is used (as the primary input) if only one name appears. In a manner analogous to the UNIX "cat" command, a lone hyphen "–" may appear within a collection of file names to show the point at which the data is taken from the standard input. This is useful where the standard input is taken from the output of a previous program, and will be used as the secondary data set, as when performing non-commutative operations such as pixel differencing.

File name conventions suggest the file extension ".im" to indicate an image file, but this choice is not mandated. If *filename* is given without any extension, then *filename.im* is the first match. Thus, in a directory containing three files { *foo.big.im, foo, foo.im* }, the file name "foo" specified on the command line finds the file *foo.im*, whereas the name "foo.big" on the command line fails to match, because a file extension is already present in the file name argument. The match mechanism searches both the current directory for "foo" and "foo.im" variants. If not successful, the search continues in the directory specified in the environment variable IMPATH. A typical variable setting might be "/u/jjjamison/images" to indicate a default repository of .im files. When an absolute path specification is given, only the file name (with a possible .im suffix) is searched; the global path variable is ignored.

1.8. Tools by Example – Digital 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. An accompanying set of figures (Plates 1 through 4) are included at the conclusion of the chapter. Two plates representative of traditional digital halftoning are also included.

Performing similar "experiments", we discovered that new techniques suggested more general forms for existing tools, leading to the creation of lower-level tools. As a result, two pieces of software (*imthresh* and even *imhalftone*) were made obsolete, as their respective functions are specific cases of more general mechanisms. Even more pleasing, the general tools are, as a rule, easier to create and easier to "performance tune" than their predecessors. This is because of their simplicity and lack of extensive special casing. The "cleverness" that handles special cases is embedded in the ways that pipe fittings are plumbed together to provide new operations where none previously existed.

Experiment #1

The unary tool *imthresh* was created to threshold a monochromatic image against some constant value (this constant is passed as a command line parameter, i.e., *-thresh {int}*). It produces binary output. The original input data is "continuous" tone artwork and cannot be printed, but is approximated by Plates 5 and 6. The thresholded tool output appears as Plate 1. The *imtomask* step reduces the fully black or white values stored in eight bits to a compact one-bit representation.

```
imthresh milkdrop.im -thresh 127 | imtomask >out
```

Experiment #2

The command line threshold parameter in Experiment #1 strongly suggested its being taken from a file. Since *imconst* generates arbitrary files of constant intensity (which can be specified on the command line) we envision thresholding as a binary tool which does a test for magnitude of its two inputs, with *imconst* used to provide a reference level for the secondary input. Thus, thresholding can be duplicated by a subtract operation, with a subsequent test to map $x>0$ values into white, else black. This last function already exists as *imtomask*. We may thus perform the test $a>b$ as $(a-b)>0$ in two steps:

```
imconst -d milkdrop.im -v 127 | imsub milkdrop | imtomask >out
```

Experiment #3

Thresholding to a uniform, constant value produces images that look (as expected) terrible, 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 as clean as possible: the 4×4 matrix of threshold weights is hard coded and the software must cyclically permute the array internally to conform to arbitrary widths and heights present in the input file. The result appears in Plate 2.

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

Experiment #4

The halftone results look good, but we need avenues of further exploration. Realizing that the "messy" internal weight table replication code is in fact a "tile" operation we had planned to implement at some point, we write *imtile*. As a bonus, the general tiler is extremely fast, and includes offset switches to be fully general. These correspond to phase shifts in the halftoning dot, a desirable feature in color digital halftoning, in which halftone screens for successive color 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 | imsub milkdrop | imtomask >out
```


Experiment #5

The 4×4 kernel, now represented as *kernel.im* in Experiment #4, 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*. Finally, the hard coded ASCII constants of *imhalftone* have found a happy niche. At this point, one can easily envision a standard sequence of operations (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
248  184   88  216
104   24   56  136
168   40    8   72
200  120  152  232
^D
```

Experiment #6

The typing of constants in Experiment #5 suggests a mechanized means to generate random numbers, and we are curious to see the appearance of such output. We note that the creation of an abstract array of random numbers (with no knowledge specific to halftoning) is all that is needed: the other code is already in place. We quickly rewrite a copy of *imconst* so that random values are substituted instead of constant values. The *-default* switch in our example borrows the dimensions and pixel specification of *milkdrop.im*, so that *imrandom* can produce output with conforming pixels. The output shows superimposed high-frequency noise [Robe62] and closely resembles the grain present in film emulsions when enlarging or "pushing" development too far (Plate 3).

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

Experiment #7

Curious as to what Gaussian distributed random numbers might provide visually, we recall that the "coin tossing" method [Kalb79] generates them through averaging small sequences of evenly distributed numbers. We decide that pixel-by-pixel averaging and summation operators are useful, and rewrite *imsub*. Rather than producing a new tool for each function needed, we fold all functions into a C-language "switch" statement, which selects the proper operation based on an external specification (the *-operation* switch). The tool is renamed *imaop* because it 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, with accompanying output (Plate 4).

```
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 #8

The brevity of most command lines and default values that usually provide accurate "guesses" is pleasing, but *imtomask* is beginning to look like a spare pipe fitting. 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 files with more than one field, something we were unwilling to write at the outset (in Experiment #1). Both *imthresh* and *imhalftone* have been made obsolete.

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

Experiment #9

Color was previously available by using *imextract* to generate three color separations, and later used to recompose them. However, both the *imtomask* and the *imthresh* step generated one-bit binary output format, making the operation cumbersome. The new *-op thresh* operator in *imaop* allows color naturally. Here our 4×4 thresholding kernel has weights for each pixel component skewed internally in a fashion analogous to a conventional printer's use of different halftone screen angles for each color separation. The color threshold table was constructed out of the monochromatic *kernel.im* using offset, crop, and merge operations.

```
imtile color4x4 -w 512 -h 512 | imaop colorim - -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 emerges, specific tools can be 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, a *-gauss* switch may be added to *imrandom*, but for current applications this is unnecessary.

As a second example, the subtract and test operation now provided by the tools may be applied to pixels containing data which do *not* represent intensity. For instance, a user encoding { *R G B Z-depth* } pixels in a depth-buffer system might wish to merge two such images. Here, the source pixel chosen is the one with the smaller

(nearer) Z value. By thresholding Z values in both files, we produce a bitmap which can subsequently be used as a mask specification to merge the two files together. Thus, the threshold operation compares values encoding the physical property of *distance*, not intensity.

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 to a goal exists, we both pare down what constitutes a minimal set and simultaneously get insights into the deep structure of the problem.

1. Plates

Plate 1 – Simple Threshold

Plate 2 – 4x4 Halftone Dot

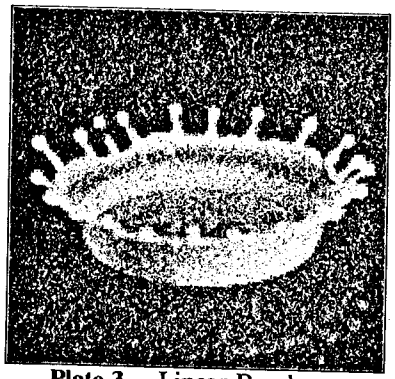


Plate 3 – Linear Random

Plate 4 – Gaussian Random

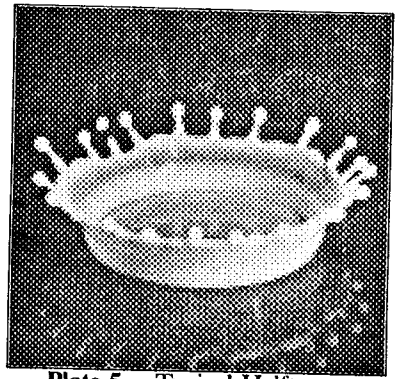


Plate 5 – Typical Halftone

Plate 6 – [Floyd75] Diffusion

Implementation

2.1. Overview

The complete image processing system is built in the C programming language upon a well-defined set of data structures. The first sections describe the data presentation formats available to the applications code, as well as the *image* data structure *per se*, which is used as a generic "handle" when dealing with raster entities, in analogous fashion to file variables.

The program code is gathered into two libraries: *imlib* and *imtool*. The first is the basic low-level library which provides all access functions to open, close, read and write raster files. The second library provides a command line parser and associated routines to allow the creation of a toolkit with a uniform user interface, error handling and other common conventions.

2.2. External Representation

The external raster representation is "beneath" the lowest code level for internal representation, as it never exists on the disk without some decoding, such as mapping header values into the associated *image* structure. Data on the disk consists of a serial stream of eight bit bytes. The stream in turn contains three parts: an ASCII header, a separator character, and the raster data proper. The stream is processed strictly sequentially, and no "padding" is mandated by the format, thus allowing for piping (serial composition of programs) of raster tools. Data should not follow the end of the data set (its final character position can be uniquely determined, so a stream EOF indication is not needed). Nor should a file end prematurely, even if the raster data be entirely zero.

The external representation is mapped into the machine by an initial call to *imheadread* or *imheadwrite* which copies the header information into the *image* data structure. This structure, described in Section 2.3.2, maintains the names of fields, size of the raster, plus internal information. The remainder of the file is mapped on a scan line by scan line basis by calls to *imread* or *imwrite* to move single scan lines between the user's data space and the external stream. Pixel-by-pixel I/O is an abstraction left to the application code.

2.2.1. Header Part

The header part consists of single line entities, separated by <cr> characters. Blank lines and lines beginning with '*' are ignored. Other lines consist of (keyword string, value string) pairs, with white space separators (<space> and <tab> in all combinations) between the keyword and value, and with white space optionally appearing before the keyword and after the value. Keywords may appear in mixed case. Mandatory keywords are the strings "width", "height" and "pixel" and optionally "compress". All save "pixel" require a digit string (positive integer) as an associated value.

The character string associated with the keyword "pixel" is of the form: <field name char><field name length>, with *char* $\in$ { `a' .. `z' } and *length* $\in$ [1..32]. Note that since the field names are unique, the format supports a maximum of twenty-six fields, and hence at most eight hundred thirty-two bits. Common examples are "r8g8b8" for twenty-four bit color images or "n1" for binary (black and white) images, with "n" (neutral) used to suggest monochromatic data. The pixel keyword defines the "bands" or "channels" which make up an aggregate pixel and gives the number of significant bits for each field. The user is free to interpret this as desired, but the conventional meaning is intensity on the range [0..1], with "background" being that unique value when all field components are zero. A sample header appears in Figure 2:

<header.im>

```
*Raster file 11 Nov 86

width  512
height 1
pixels r8g8b8
compress 3
**raster data follows **
```

Figure 2 – Raster File Header

2.2.2. Separator

The separator character is <ff> (control-L) which conventionally serves as a page break character in ASCII. Thus, standard tools, such as the UNIX *more* command will pause when attempting to display the header information beyond this separator. Because the header specification allows blank lines, the <ff> may appear on a line of its own. Note that the raster part immediately follows the <ff> character, with no trailing <cr>, because all ASCII values (including <cr>) may appear within the raster data part.

2.2.3. Raster Proper

The raster proper consists of a byte stream with each byte taken to represent an unsigned, eight-bit integer. These in turn may be packed to form integers of larger significance to represent field values. This packing is described in Section 2.3.1.3. Because the construction of integers is defined by the presentation layer code (and not by the host machine's hardware byte order), the code is fully portable. In simple cases the mapping is trivial. For instance a 640×480 file of "r8g8b8" twenty-four bit color values is stored on disk as 640×480×3 bytes, beginning with the red intensity value of the upper left-hand corner of the image and ending with the blue intensity value of the lower right-hand corner. This left-handed coordinate system (y increases as one moves down the screen) is typical of raster-oriented software, but is unlike the right-handed world coordinates typically used elsewhere in computer graphics [Newm79]. When compression or fields with other than eight bit precision are used, the external representation is less obvious, but still gives a unique representation.

2.3. Internal Presentation

The raster library (*imlib*) is constructed as a layering of data presentation levels, each of which guarantee a consistent presentation format to the applications code, regardless of machine specifics. The lowest presentation level merely copies data to or from the external media, while the highest level provides normalized floating point values for data appearing in each pixel field. Intermediate levels provide for the conversion of byte orders or the unpacking of pixels into aggregate fields.

Data presented to the application code is most often in the form of (unsigned) thirty-two bit integer arrays. Based on presentation level, each integer element may contain a collection of pixels, a single pixel, or a single pixel field component. A provision is also made to store thirty-two bit floating point values externally, though the external files may then become non-portable. A presentation layer assures consistent byte ordering when forming machine integers, so the "Big and Little Endian" problem [Cohe81] does not exist.

2.3.1. Presentation Layers/Levels

The level names for each section following are in fact the *#define* names passed as parameters to the routines *imread* or *imwrite* to specify what level of presentation is required by the application. The layers are summarized in Table 3 and described in the following subsections, in top-down order.

LEVEL	NAME	INPUT	OUTPUT
6	IMFLOAT	imfbuffloat.c	imfbuffix.c
5	IMFIELD	imbufextract.c	imbufinsert.c
4	IMPIXEL	imbufunrep.c	imbufrep.c
3	IMFLIP	imsysflip.c	imsysflip.c
2	IMPACK	imbufunpack.c	imbufpack.c
1	IMRUNCODE	imbufdecode.c	imbufencode.c
0	IMRAW	imread.c	imwrite.c

Table 3 – Data Presentation Levels

2.3.1.1. IMFLOAT

The scan line data is a parallel set of arrays containing floating point field values, one array per field. Thus, the data is broken down into distinct field components. Array elements are floating (thirty-two bit single precision) values on the range [0.0..1.0]. The *ith* field array is specified by *image->fbuf[i]*, where $0 \le i <$ *image->fields*. This array has *image->width* elements. For each field, the first element (a fully qualified machine integer) of the current scan line is *image->fbuf[i][0]*, the last is *image->fbuf[i][image->width-1]*.

This level is useful when performing mathematical evaluations on files independent of their input precision, particularly when performing binary operations on two input streams of differing precision. An example appears as <imbw.c> in Section 2.4.6.

2.3.1.2. IMFIELD

The scan line data is a parallel set of arrays containing unsigned integer values, one array per field. Thus, the data is broken down into distinct field components. Each element is an unsigned, thirty-two bit integer, containing j bits of field precision, with the left-most bits zeroed. Fields are no larger than thirty-two-bit precision, nor do they span machine integers. The *ith* field array is specified by *image–>fbuf[i]*, where $0 \le i <$ *image–>fields*. This array has *image–>width* elements, and an element precision (j) of *image–>flen[i]* bits.

This level is most commonly used to perform integer arithmetic and logical operations on fields, particularly where the operations will not become ill-conditioned when evaluated in a fixed-point domain. An example <imblend.c> appears as Figure 7 in Section 2.5.6.

2.3.1.3. IMPIXEL

The scan line data is in a single unsigned integer array. The individual fields are packed into contiguous integers in this array; fields may not span integers. This level is useful for dealing with aggregate pixel quantities represented by some multiple of integers. For instance, the tool *imcrop* copies pixels without regard to the component values, while maximizing the item size (machine integers) used to perform the copy. The value *image–>pixwords* defines how many contiguous words form each pixel, beginning at *image–>buf[0]*.

To illustrate the packing scheme, consider the representation "a10b20c10". At the IMFIELD level, *image–>fbuf[0][0]* is a 10-bit unsigned integer representing the 'a' field of the leftmost pixel on the current scan line. At the IMPIXEL level, the "a" field occupies bits <0..9> (in our notation, bit n has weight 2^n; bit 0 is the lsb), "b" occupies <10..29>, and there is not enough room left in "c", which must occupy bits <0..9> of the next machine word and *image–>pixwords* is therefore two. The leftmost pixel is thus contained in *image–>buf[0]* and *image–>buf[1]*. Note that "a1b32" and 'b32a1' would both require two machine words for storage, with the former filling the second word to bit <31>, whereas the latter fills the second machine word to bit <0>.

The routines guarantee that non-zero residue in undefined portions of the machine integers are zeroed both on reading and on writing, making it impossible to hide data within the format. This also allows for faster testing of background pixels (those with all fields identically zero), as the comparisons may be done on a by-machine-word basis. An example <imhalf.c> appears as Figure 8 in Section 2.5.6.

2.3.1.4. IMFLIP

Data is thirty-two bit integers, possibly with several pixels per word or possibly with several words per pixel. The "words" are in the natural arithmetic word arrangement of the host machine. No pixels (nor their internal fields) span word boundaries, but pixels of less than sixteen bits length may be packed to fill a single machine integer. The packing is in arithmetic order beginning with the least significant portion of the machine integer.

As a consequence of this packing, all submultiples of thirty-two achieve 100% storage efficiency and most common cases achieve >88% efficiency. Worst-case storage utilization occurs when the final bit in a pixel must allocate an otherwise empty byte. The seven unused bits comprise the largest lost fragment with "n17" data, which must encode in three bytes: shorter lengths allow packing and longer lengths minimize the loss of seven bits. The worst-case lower bound curve appears in Figure 1.

2.3.1.5. IMPACK

Data is effectively a byte stream, whose order does not necessarily reflect the machine's integer conventions (though it does on the VAX). Bytes are unpacked, so that a leftmost "r8g8b8" pixel would form the first three bytes of the thirty-two bit field in *image–>buf[0..2]*, cast as a character array, with *image–>buf[3]* zero. This is an intermediate level and is not usually used directly.

2.3.1.6. IMRUNCODE

At this level, whole bytes that are extraneous have been thrown away. For example, if the data format was "r8g8b8", it is known that every fourth byte is zero, since only twenty-four bits of thirty-two are used. The algorithm compacts the data buffer in place, removing useless bytes. Additionally, pixel sizes of eleven and twelve bits pack to form pixel pairs of twenty-two and twenty-four bits, which may also be encoded in only three bytes per machine integer. Since the byte order has been standardized at the IMPACK level, the useless bytes are always located in the same place, regardless of machine. The data presented at this level is not compressed (run coded).

2.3.1.7. IMRAW

If run coding is not in effect, the data is in a form ready to be transferred to disk. If coding is enabled ("compress" keyword has a non-zero value n, as recorded in *image–>cbytes*), then the stream is in the coded form described: the stream is blocked into "groups" of length n bytes and run-length coded on the basis of identical groups. An additional byte is appended to each run of n bytes, defining from 0..255 additional repetitions of that group. Longer runs give rise to sequences of identical groups. Note that group boundaries do not necessarily correspond with pixel boundaries. For instance, run encoding of binary (one bit) data will compress file size on the basis of matching eight bit patterns in consecutive order, not on the basis of black or white runs in the data (which would be less efficient). The number of runs on any given scan line is written as a four byte integer in lsb order prefacing that scan line. This allows the contents of an entire scan line to discarded once this count has been read, without enumerating runs along the scanline in an effort to recover the beginning of the next consecutive row. This value is recorded in *image–>runs* on a per scan line basis.

Run length groups might not divide the width of the scan line integrally. In this case, the last "runt" group is expanded to form a full length group, which will necessarily never match the previous group, and will have a replication count of zero. Thus, the number of bytes to transfer can always be taken by multiplying *image–>runs* by *image–>cbytes+1*.

Data presentation below this level appears on the external stream only. The I/O routines (plus packing conventions) limit the number of bytes transferred to a minimum. As a consequence, a $W \times H$ sized raster of J components, each of byte length, stores in $W \times H \times J$ bytes, exactly, assuming no compression is in effect.

2.3.2. Description of the *image* structure

Each *image* structure owns both its physical file and all of its I/O and pixel buffers. This minimizes the parameters used in procedure calls, as only the image object is passed in many cases. As a consequence, scan line buffers are not user supplied. This is also a convenience, but implies that data must be copied into the output structure; two image entities should not share (through use of pointer values) the lower-level buffers, e.g., *image–>buf* or *image–>cbuf*. The penalty is that the machine instruction savings for "b[i] += 2" versus "a[i] = b[i] + 2" are lost, but this not significant. The *image–>fbuf* field level data buffers may be shared where operations such as field extraction and merging take place, but this is not generally advisable.

Additional fields in the *image* structure are used internally to define shift and masking offsets used to (un)pack and (de)replicate data when it is to be presented at various levels.

2.4. IMLIB Code Description

The "im" software consists of a body of roughly thirty routines, typically stored one procedure per file of the same name. The routines employ no recursion and maintain all state within an *image* structure, which exists on a one-to-one basis with any raster stream being read or written, so that no global state is needed. The file *im.h* is the sole definitions file for this and other structures. The remaining routines are named *imxxxyyy.c* where *xxx* is the optional direct object receiving the action defined by *yyy*. For instance, *imbufpack* is the routine used to pack the contents of *image–>buf*. File names are limited to fourteen characters to maintain compatibility across C compilers. The routines can be grouped as shown in Table 4.

The routines are grouped into a master library *imlib* which must be linked in with any imaging software. This library is build from the C routines. The include file *im.h* must additionally be referenced by any application code.

2.4.1. Open/Close (Header) Routines

The routines *imheadread* and *imheadwrite* are used for opening image files on an existing, open file stream. The companion *imheadfree* is used at the conclusion of the file; it additionally closes the associated stream, as no data should appear after the raster data. The latter also returns all buffers and the structure itself to the system storage pool. On writing, an image can be allocated using the macro *imheadnew*, and relevant raster specifications filled in, as described below. Alternately, a third routine, *imheadclone* allocates a new, duplicate copy of an *image* object. The user should then ascribe a new stream to it.

Fields corresponding to the raster keywords must be filled in before an object can be used (on writing). Simple pointer assignment cannot be made in the case of the pixel string, because deletion of an *image* object will attempt to free an unallocated

Buffer	Routines Header	Other
imbufdecode	imheadalloc	imchar
imbufencode	imheadbuild	imcopy
imbufextract	**imheadclone**	**imerr**
imbuffix	**imheadfree**	**imfieldmap**
imbuffloat	**imheadinsert**	imlookupkey
imbufinsert	imheadparse	imsysaux
imbufmask	**imheadread**	imsysfld
imbufpack	imheadtable	**imread**
imbufrep	**imheadwrite**	imreadspec
imbufunpack		**imwrite**
imbufunrep		imwritespec

(user entry points in **bold**)

Table 4 – IMLIB Routines

pixel string. Instead, the routine *imheadinsert* is passed an object, keyword and string value of the entry to be filled in. This mechanism is also useful in assigning the raster width or height if they are present in a string format. The keyword names are listed in Table 5; code of this nature appears in Figures 4 and 5.

A subsequent call to *imheadwrite* will implicitly complete the internal specification of the *image* entry and allocate the physical buffers prior to writing the stream header. On reading, *imheadread* is passed a file stream and an *image* object is allocated, filled in appropriately, and returned. After reading or writing, the stream is left positioned at the first scan line.

2.4.2. Scan Line Read/Write Routines

In typical use, two principle routines are used to read or write an image file. Additionally, a number of fields within the *image* structure are used to record (or recover) spatial and pixel attributes of an image file. The routines *imread* and *imwrite* take as parameters an active *image* object, plus an associated presentation level, as described earlier. Their use is best described by example. The segment <imskip.c> opens an image file, reads (and discards) its contents and closes it. We choose the lowest presentation level, as scan line contents are not important:

<imskip.c>

```
#include "im.h"
#define STDIN 0

image in;
int row;

in = imheadread(STDIN);
/*
 * discard entire file
 */
for (row=0; row<in->height; row++) imread(in, IMRAW);
imclose(in);
```

Figure 3 – Image File Reading

On open, the *image* structure will record the header values and derive other useful values. For instance, a call to *imheadread* when passed a stream for an uncompressed 128×256 twenty-four bit RGB image would form:

image–>	value	keyword
width	128	WIDTH
height	256	HEIGHT
pstr	"r8g8b8"	PIXELS
cbytes	0	COMPRS
fields	3	
fname[0..2]	['r', 'g', 'b']	
flen[0..2]	[8, 8, 8]	
buf	0:R:G:B[0..127]	
fbuf[0]	R[0..127]	
fbuf[1]	G[0..127]	
fbuf[2]	B[0..127]	

Table 5 – Sample Image Object

The *cbytes* field indicates no run length encoding is present on the external stream, but this is not important for presentation levels above IMPACK. The **keyword** column indicates the constant name passed as a second parameter to *imheadinsert* when building a header specification from string values.

When writing, a new *image* record must be created and the first three values filled in, thus defining the new raster. The code segment <imblack.c> will create a "black" image:

<imblack.c>

```
image out;
int row;

out = imheadnew();
imheadinsert(out, PIXELS, "r8g8b8");
out->width = 128;
out->height = 256;
out->cbytes = 0;
imheadwrite(out); /* completes and writes header info */
for (row=0; row<out->height; row++) imwrite(out, IMRAW);
imclose(out);
```

Figure 4 – Image File Writing

In practice, output images are typically formed with spatial or pixel attributes matching some other input file, suggesting that the *image* structure fields need not be filled in on an entry-by-entry basis. A standard tool-building library (described in Section 2.5) provides this function automatically.

2.4.3. Field Identification

In practice, generic tools will use all fields present in a pixel, whereas specific tools may require certain fields to appear. By convention, the field names *R G B N* define *red green blue neutral* spectral components, respectively. The routine *imfieldmap* may be passed an *image* and character string of requisite fields. The routine returns the number of matching fields *n* based on the set of field characters. It also sets the entries *image–>factive[0..n-1]* with integers giving the corresponding location of the sought field. At run time, fields which are not active will not be extracted above the IMPIXEL presentation level.

For example, the call *imfieldmap(image, "GZ")* with *image* a raster of pixel type "r8g8b8z8" would return the value two (because both fields were present), and calls of the form *imread(in, IMFIELD)* would fill in the field arrays in–>fbuf[1], and in–>fbuf[3] only. Additionally, the first two elements of *in–>factive* would be {1,3}, as an aid to the program locating the *fbuf* indices of the sought fields. This is depicted in Section 2.4.6.

2.4.4. Error Handling

The *im* routines called by the application normally return a zero (normal completion) code. Error conditions detected by the low level routines always pass through the internal routine *imerr* and control is then returned to the caller, with a non-zero return code. This prevents the *im* package from causing program termination when used in a larger setting, as when used as a file storage mechanism in an interactive imaging system (such as Palette [[Higg86]]).

When *imerr* receives control after error detection, it will check for a non-zero procedure variable, *_imerr*, and if found, pass control to this routine with the error string. Thus, stand-alone software using *imlib* (such as the *imtool* tool building library) may assign this variable the name of a routine which posts the error message on the system error stream, and then abort, with control never returning from *imerr*. This also appears in <imbw.c>.

2.4.5. Other Routines

The files *imsysfld.c* and *imsysaux.c* contain heavily optimized routines to provide the lowest level of data manipulation. The first contains procedures to extract and insert integers of variable bit precision into standard (thirty-two bit, unsigned) machine words. Hand optimization of the C code for the VAX led to object code which ran at speeds comparable to hand-written VAX assembler, despite the VAX architecture supporting "insert" and "extract" machine instructions, in exactly the form required. This is consistent with findings on other hardware, as advocates of the RISC machine architecture have pointed out [Patt85].

The second file contains routines to zero and copy the contents of integer arrays (which in many UNIX systems is available within the *bstring* library). A final procedure set checks machine integer byte ordering at run-time, and provides an integer reordering routine, used by the data presentation levels. Four orderings (their are 4!=24 possible) are supported, which is believed adequate for any machine architecture in existence. System byte order is maintained in the global static *_imbyteorder*. This variable is set and the conversion routine called implicitly on the first and subsequent calls to *imread* or *imwrite* which require knowledge of byte order.

2.4.6. Code Example

A functional example <imbw.c> which converts RGB files of arbitrary precision into eight-bit monochromatic files is presented in Figure 5. The tool operates independently of the input file's field precisions, spatial dimensions or the presence of additional (non-RGB) fields. The output file conforms in spatial dimension and has identical run encoding to the input file, though this is normally invisible. The algorithm uses the RCA devised NTSC [Limb77] standardized system for broadcast video.

2.4.7. Internal Code Optimizations

Three global "flags" are defined in *im.h* and used internally throughout the library.

The REGISTER macro is normally defined as "register". It typically appears as a "REGISTER int *buf, count" statement within a routine's innermost loop and achieves some increase in speed by forcing certain variables into registers. As a rule, no more than six registers are used (in addition to any potentially ascribed to the formal parameter list outside the procedure block). This is a hand-coded means to perform global register optimization. Their use can be overridden by setting REGISTER to the null string.

The FASTLOOP macro prefaces the beginning of each innermost loop. Besides visually highlighting the location of such loops, FASTLOOP (on the VAX) uses a compiler directive to force the body of the loop to begin on a machine doubleword (octabyte) boundary. This allows the instruction execution pipeline to fill the eight byte pre-fetch cache the least number of times. Unfortunately, most VAX/UNIX object loaders accept alignment directives only to the nearest quadbyte, so half the time we force code to be centered within this pipeline. It is hoped that this will be remedied someday. Other systems may wish to find suitable equivalents, or set the macro to null, without incurring a significant loss of performance.

<imbw.c>

```
/*
 * imbw.c -- Convert a file with RGB (+extra) fields to 8-bit b/w.
 *           Any precision ok, because of IMFLOAT presentation.
 */

#include <stdio.h>
#include "im.h"
#define STDIN 0         /* low-level stream handles */
#define STDOUT 1
#define RED 0.30        /* weighting coefficients for dot product */
#define GRN 0.59
#define BLU 0.11

fail(str)
    char *str;
    {
    fprintf(stderr, "imbw failure -- %s", str); exit( 1 );
    }

main()
    {
    image imin, imout;
    int row, col;
    float *rbase, *gbase, *bbase, *nbase;
/*
 * input from standard input, check for all three fields present
 */
    _imerr = fail;                      /* intercept package errors */
    imin = imheadread(STDIN);
    if (imfieldmap(imin, "RGB") != 3) imerr("no R,G,B fields");
/*
 * output on stdout, with matching spatial dimensions but 8-bit b/w
 */
    imout = imheadclone(imin);
    imheadinsert(imout, PIXELS, "n8");
    imout->file = STDOUT;
    imheadwrite(imout);
/*
 * locate the input and output buffers
 */
    rbase = (float *)(imin->fbuf[imin->factive[0]]);
    gbase = (float *)(imin->fbuf[imin->factive[1]]);
    bbase = (float *)(imin->fbuf[imin->factive[2]]);
    nbase = (float *)(imout->fbuf[0]);
/*
 * CONVERSION LOOP: use the YIQ North American Broadcast Convention
 */
    for(row=0; row<imout->height; row++) {
        imread(imin, IMFLOAT);
        for (col=0; col<imout->width; col++) {
            nbase[col] =
                RED*rbase[col] + GRN*gbase[col] + BLU*bbase[col]; }
        imwrite(imout, IMFLOAT);
        }
/*
 * free storage and quit
 */
    imheadfree(imin);
    imheadfree(imout);
    exit( 0 );
    }
```

Figure 5 – Stand-alone Image Filter

The *image* fields *onebuf* and *bytefields* drive the execution of special case code. In the first case, pixel or field extraction are identical at the IMPIXEL or IMFIELD levels, and the redundant step can be removed in the presentation hierarchy. In this case, the fields *image->buf* and *image->fbuf[0]* are pointers of identical value. The *bytefields* flag is true if all fields are of eight bit length. In this case, operation to the IMFIELD level (or as above, to the IMPIXEL level in the case of one field) are treated as special cases. Normally, byte unpacking of the contents of *image->buf* is done to the IMPIXEL level by using bit packing operations, followed by any field extraction. In the special case, bytes from the input stream are copied directly to the *image->fbuf* field arrays. This step additionally bypasses any system byte reordering, as the byte copying operation is not arithmetic in nature. This step is invisible to a caller of *imread* or *imwrite*. The general case code is implicitly reentered above this presentation level, should the application code should require floating point presentation of byte data.

2.5. IMTOOL Code Description

The *imtool* library provides a uniform mechanism for consistent tool creation. This interface standardizes conventions of file and command line specification, error reporting and tool identification. The routines also reduce the amount of code needed by grouping the tools by class (see Section 1.6.2) and providing default input and output *image* streams, through a body of common library code.

2.5.1. Tool Set Up

Tools are set up by an initial call to *imopen* with parameters passed to specify the given tool name, operation class, plus the system provided *argc* and *argv* structures, in that order. The given name is used to identify the running code when any tool is passed the *-usage* switch, or when an error occurs. The *argc,argv* values are saved in global variables, and are subsequently used by a simple command line interpreter. The *class* value is a *#define* constant, taken from *imtool.h*, with values and actions defined in Table 6:

Tool Class	Pri In	Aux In	Std In	Std Out	Comments
IMNONE					tool interface, but no rasters
IMSOURCE				●	default -pixel -width -height
IMDRAIN	●		●		no output
IMUNARY	●		●	●	imout matches imin
IMBINARY	●	●	●	●	imout matches imin
IMTRIPLE	●	●	●	●	adds imaux2 as tertiary input
IMMULTIPLE				●	I/O images in _imfilev[]

Table 6 – Tool Class

On return from *imopen*, the global structure *_imfilev[]* has elements filled in with *image* objects, named on the command line (see Section 1.7). The command line parser recognizes raster files as tokens representing valid file names, preceding the first command line switch token (which begins with a hyphen). This is in contrast

to most UNIX filters. The number of input files found is stored in *_imfilec*, in a manner analogous to *argc* and *argv*. One output stream (for non-IMDRAIN tools) is allocated on the standard output; it is located in *_imfilev[0]*. The tools library *#include* file *imtool.h* provides macro names *imout*, *imin*, *imaux* and *imaux2* as shorthand means to specify the first four elements of this vector.

The tools marked "Std In" in Table 6 require an exact number of input files. Should the command line supply one less file name than the required number, the standard input will be read to make up the required input. In this case, a lone hyphen "-" may be placed in the command line to locate the standard input elsewhere in the *_imfilev[]* image stream vector (see also Section 1.7.3). This is useful in sequences where tool operation does not commute.

After a call to *imtool* all input streams are opened for reading. The output image (excepting IMDRAIN tools) is referenced by *imout*, which is an *image* stream on the standard output. The output header is not written, as tools may wish to redefine the pending raster width, height, pixel or compression specifications, which normally default to the attributes of the primary input. For IMSOURCE files, the output specifications default to a 512×512 "r8g8b8" image.

2.5.2. Command Line Specification

Immediately following the *imopen* statments are optional calls to define valid command line switches. Each routine is called with a switch name string. A second parameter is often included to provide a default value. The routines return either the default value, or the command line value, if found. The boolean type returned is a (non)-zero integer. The *table* type is an array of up to twenty-six integers, with default length and pixel precision taken from an *image* raster specification, also passed. These routines are summarized in the following table:

Type	Name	Second Parameter
boolean	imflagany	(none)
string	imflagstr	default string
integer	imflagint	default integer
float	imflagfloat	default float
table	imflagtab	default *image*

Table 7 – Command Line Switch Routines

Following the last switch, a mandatory call to *imflagcheck* is made to complete the tool specification section. This routine checks for any unclaimed switch tokens on the command line. Additionally, this routine provides an early return point on tool invocation under the *-usage* switch. As a consequence, no code which calls error routines or otherwise generates output should appear before the call to *imflagcheck*. A model tool might thus resemble:

<immodel.c>

```
    int swI, swR;
    imopen("immodel", IMBINARY, argc, argv);
    swI = imflagany("invert");
    swR = imflagint("repeat", 20);
/*  additional switch code */
    imflagcheck();          /* end of command specs */
    imheadwrite(imout);     /* complete output spec */
/*
 * APPLICATION CODE (by scan line)
 */
    imclose();
    exit( 0 );
```

Figure 6 – Model Tool Layout

2.5.3. Error Routines

After tool opening, all system error messages are routed to the standard error stream through a mechanism described in Section 2.4.4. Tool errors cause the program to quit with a non-zero exit code; there are no warning errors. Error messages are prefaced with the name and class of the running tool; this is helpful when many tools are run concurrently. Error output may be squelched under the *-quiet* switch, common to all tools.

User specified terminal errors may also generate text by using the system internal error entry points; these routines are in fact part of *imlib*. A multiplicity of error routine names are provided to allow strong type checking in systems such as *lint*. The first parameter to each routine is the error text string. A possible second parameter is a typed value, with matching *printf* escape characters embedded within the first parameter string. The routines are tabulated below:

Routine Name	Second Parameter	Printf Escape
imerr	(none)	
imerrint	integer	%d, %u
imerrstr	string	%s
imerrchar	char	%c
imerrfloat	float	%e, %f, %g

Table 8 – Error Routines

2.5.4. Tool Closing

The program finishes with a call to *imclose*, which discards all active *image* structures presently in use, and closes the associated streams. The user should subsequently return program control to the operating system with a zero exit code.

2.5.5. Other Routines

The file *imtoolaux.c* provides two high-speed data routines particularly useful for dealing with one bit data. The routine *imbytereverse* reverses the bit order within bytes and optionally complements the bits as well. This is useful because many binary output devices consider '1' to represent black, and order the significant bits from msb to lsb ("left to right"). The tools *imheadon* and *imheadoff* use this feature. The routine *imbinpack* maps integer arrays of (non)-zero data (such as data at presentation levels IMFIELD or IMPIXEL) into packed binary strings. These routines are used by *imversw* and *imcanw* to form a binary output raster when given input of arbitrary pixels.

2.5.6. Code Examples

Two complete examples are provided. The binary tool *imblend* is used to cross-fade between two images. For the sake of simplicity, the input streams must match in both pixel and spatial dimensions, and additionally by matching order of field names. The example invokes all routines typically used.

The final tool *imhalf* "down-sizes" images by a factor of two, by performing a nearest-neighbor resampling. It provides a function identical to

```
imsample -s 0.5 -x 0 -y 0
```

while lacking the advantage of full generality. This tool preserves pixel atomicity, and thus deals with the data at the lowest presentation level conventionally used (IMPIXEL). It additionally makes extensive use of register variables and nests program loops by order of repetition length, thus giving a representative model of hand-tuned, high-speed tools.

<imblend.c>

```
/*
 * imblend.c -- Cross-fade of "[1-fade] * pri + [fade] * sec" input.
 *              Note: raster size, field names and order must match.
 */

#include "im.h"
#include "imtool.h"

main(argc,argv)
    char **argv;
    {
    float swF;
    int row, fld;

    imopen("imblend", IMBINARY, argc, argv);
    swF = imflagfloat("fade", 0.5);       /* default 50/50 blend */
    imflagcheck();
/*
 * error checks for switch values, spatial and pixel conformality
 */
    if ((swF>1.0)||(swF<0.0))imerrfloat("fade %f out of range",swF);
    if (imin->width != imaux->width) imerr("widths mismatch");
    if (imin->height != imaux->height) imerr("heights mismatch");
    for(fld=0; fld<imin->fields; fld++)
        if (imin->fname[fld] != imaux->fname[fld])
            imerrint("non-matching field names in field #%d",fld+1);
/*
 * now generate output with pixel type and size conforming to input
 */
    imheadwrite(imout);
/*
 * do the binary operation
 */
    for(row=0; row<imout->height; row++) {
        imread(imin, IMFLOAT);
        imread(imaux, IMFLOAT);
        for(fld=0; fld<imin->fields; fld++) {
            REGISTER int count;
            REGISTER float *inl, *inr, *out, lmix, rmix;
            inl = (float*)(imin->fbuf[fld]);
            inr = (float*)(imaux->fbuf[fld]);
            out = (float*)(imout->fbuf[fld]);
            lmix = swF;
            rmix = (1.0 - swF);
            count = imout->width;
/*
 * scan line (field) inner loop
 */
            FASTLOOP
            do  {
                *out++ = (*inl++)*lmix + (*inr++)*rmix;
                } while(--count > 0);
            }
        imwrite(imout, IMFLOAT);
        }
/*
 * finish
 */
    imclose();
    exit( 0 );
    }
```

Figure 7 – Binary Tool

<imhalf.c>

```
/*
 * imhalf.c -- Scale down arbitrary raster files by 2:1.
 *             Note: round size, so zero-size output is avoided.
 */

#include "im.h"
#include "imtool.h"

main(argc,argv)
    char **argv;
    {
    int row, words;

    imopen("imhalf", IMUNARY, argc, argv);
    imflagcheck();
/*
 * overwrite the output raster parameters (taken from imin)
 */
    imout->width  = (imout->width+1)/2;
    imout->height = (imout->height+1)/2;
    imheadwrite(imout);
/*
 * for each scan line
 */
    for(row=0; row<imin->height; row++) {
        if (row&0x1) imread(imin, IMRAW);  /* discard odd lines  */
        else {
            imread(imin, IMPIXEL);         /* process even lines */
/*
 * for each integer which might represent a (partial) pixel
 */
            for(words=0; words<imout->pixwords; words++)
                {
                REGISTER int count;
                REGISTER int32 *ibase, *obase, iskip, oskip;
                count = imout->width;
                oskip =  imout->pixwords;
                iskip =  2*oskip;
                ibase = &(imin->buf[words])-iskip;  /* predecr  */
                obase = &(imout->buf[words])-oskip; /* for loop */
                FASTLOOP
/*
 * for each (partial) pixel location along the scan line
 */
                do  {
                    *(obase += oskip) = *(ibase += iskip);
                    } while (--count > 0);
                }
            imwrite(imout, IMPIXEL);
            }
        }
/*
 * finish
 */
    imclose();
    exit( 0 );
    }
```

Figure 8 – Unary Atomic Tool

Tools Description

Table 9 provides an overview of the two dozen tools presently in existence, using our taxonomy. Following the table is a synoptic description of each tool, together with one or more examples which demonstrate the tool's use, particularly when used as part of a more general synthesis step.

Historically, the level 0 (and some level 1) source, drain and unary tools were the first written. This set is of particular importance, because they may all be joined by the UNIX pipe facility, with no multiple inputs required and with no member requiring a large amount of storage (as would be true for raster rotation). In this scenario, the source and drain tools correspond to the leftmost and rightmost tools appearing on a shell command line. Also present in our table are a number of tools which require no switches at all. These are typically highly-specific user written functions. Their presence indicates that an all-embracing higher-level tool has not been discovered which could possible subsume their functions into a generic one.

3.1. Imaop, Imlop – Arithmetic or Logical Binary Operations

Imaop performs standard arithmetic operations on pairs of pixel fields, with these pairs taken from matching pixels in the primary and secondary inputs. Imlop additionally performs bitwise logical operations on pixel pairs and thus requires that both pixel names and bit precisions conform. The operations supported are specified as an argument string to the mandatory *-operation* switch; these operations are defined in Table 10. Where operations are not commutative and the standard input forms an operand, the variant

```
imaop primary.im - -operation ...
```

allows the standard input to become the secondary file. This tool embraces all unary and binary arithmetic functions when coupled with *imconst*. The output file conforms in size to the primary input and the secondary file may be offset relative to (but within the bounds of) the primary file by using the *-xoffset* and *-yoffset* switches. In this case, regions of primary input with no paired secondary input pixels are copied verbatim to the output. The present incarnation insists that pixel specifications conform exactly and that all pixel fields be called into play. Alternately, if either raster contains a pixel with only one field, its value is replicated as needed to pair with the fields present in the other input stream.

For each operation, the field integers are taken as unsigned, positive values and the result of the arithmetic or logical operation is potentially "clipped" to lie within the range $[0..2^n-1]$ with n the number of significant bits for that field. The operation "division by zero" results in zero as output.

Examples:

```
imaop face1 face2 -op aver >out.im          # blend two rasters
imconst -d pic | imlop pic -op xor >picinv.im   # invert pic.im
```


NAME	LEVEL	CLASS	SWITCHES			
			x,y	w,h	p,v	other
aop	0	B	•			operation
box	1	U		•		bitincrease
canw	0	D	•			spool copies invert transp magnify
check	0	D				
const	0	S		•	•	
crop	0	U	•	•		
cspace	0	U				space decode
degasr	0	U				mapoff
expr	0	U				pixel expression
extract	0	M				fields
flatten	0	U				
floyd1	1	U				
halftone	1	B				
headoff	0	D				invert bytereverse
headon	0	S		•	•	invert bytereverse skip
hist	0	U				aggregate
ikr	0	S	•	•	•	shift res frame noframe
ikw	0	D	•			shift res zoom gamma aspect...
irw	0	D	•			gamma mapoff
kern	1	B				normaloff offset
linear	1	U		•		scale power
lop	0	B	•			operation
map	0	U				
margin	0	U				margin overlap top bottom left right
median3	1	U				
ramp	0	S		•	•	
random	0	S		•	•	maxval seed
rect	0	D				
rotate	2	U				transpose mirror flip
rtr	0	S	•	•	•	device
rtw	0	D	•	•		device gamma mapset
sample	0	U	•	•		scale
shear	0	U				shear
tabin	0	S		•	•	
tabout	0	D				align hex
tile	2	U	•	•		
tobw	0	U				
tocolor	0	U				
tofont	0	D				
tomask	0	U				
unmap	1	B				
versw	0	D	•			invert spool
xw	0	D				maxvals bluebias redbias...

Table 9 – Toolkit Taxonomy

OP NAME	EXPR	PROPERTIES		
		commutes	range clip	imlop only
add	a + b	●	●	
and	a & b	●		●
average	(a+b+1)/2	●		
clear	a &! b			●
dif	‖ a - b ‖	●		
div	a / b		●	
max	(a>b) ? a:b	●		
min	(a<b) ? a:b	●		
mul	a x b	●	●	
on	b			
or	a | b	●		●
over	(b) ? b:a			
sub	a – b		●	
thresh	(a>b)?1:0			
under	(a) ? a:b			
xnor	a ^! b	●		●
xor	a ^ b	●		●

Table 10 – Function Table for *imaop* and *imlop*

3.2. Imbox – Box Filter Function

Imbox provides a box filter function which is useful for scaling images down by integrally smaller sizes. Scaling to larger sizes must use other techniques, such as *imlinear*. The output pixels are formed as the sum of all input pixels within a sampling box of dimension specified by the switches *-width* and *-height*. These are integer values; pixels with fractional coverage are not treated. The output file has values normalized by the number of pixels contained in the filter box (unweighted averaging). The output file has identical pixel specification as the input, but in some cases it is desirable to increase the bit significance of all fields with the *-bitincrease* switch, particularly when the input is binary data. The present implementation allows special case testing for box widths of four or eight pixels with binary (one bit) input data, by performing byte operations to give a large speed-up. This is particularly useful in converting high resolution bitmap rasters (such as raw Versatec files) into rasters of lower spatial resolution but with higher pixel precision.

Example:

```
imheadon <v.bits -b -w 2112 -h 1700 -c 0 -p n1 |\   # get bits,
imbox -w 4 -h 3 -b 7 | imikw                        # filter and display
```


3.3. Imcanw – Imagen/Canon Output

Imcanw formats "im" files into the Imagen *impress* language for subsequent laser printing. The output is sent to the standard output. Input data is typically "n1" but generally, any pixel field named "n" or "y" (failing this, then the first pixel field) is tested for zero/non-zero value, in any precision. Zero prints as black (mark), non-zero prints as white (no mark). The *-xoffset* and *-yoffset* switches are used to offset the location of the data; these generate *impress* absolute positioning commands which preface the output bitmap. The offset values are in machine units, which for most Imagen class devices is three hundred dots per inch.

A number of additional switches are available. The *-invert* switch reverses the sense of white and black. The *-transparent* flag sets an *impress* directive allowing page entities beneath non-black printing areas to appear, the default being opaque. The *-magnifypow* switch allows scaling by 1×, 2× and 4× for the current page. The *-spool* switch encapsulates the *impress* fragment with an appropriate single page *impress* header, allowing direct file transmission to the Imagen printer, else the fragment must be included within an *impress* document through other means. When spooling, the *-copies* switch sets a field in the encapsulating header, allowing for multiple one page copies with only one file transmission.

Example:

```
imfloyd1 grey8 | imcanw -s | lpr -Pim # h'tone format and queue
```

3.4. Imcheck – Consistency and Range Check

Imcheck scans over an entire "im" file, then reports bit use and maximum and minimum values seen in the image on a global, field-by-field basis. In addition, it reports (if non-zero) the number of extra trailing characters present in the "im" file. It is particularly useful in checking over raster data files of unknown format. The latter step checks that proper raster conformance was achieved. The bit use summary detail often gives some clue as to the significance of the data. For instance, six bit scanned data shifted up to eight bits will show two zeros in the lsb; min and max figures indicate how well the data spans the available range. The tool is also useful in performing quick checks to new software systems, where the possibility of bit drop-out is present in the newly created output.

Example:

```
imikr -x 100 -y 200 -w 4 -h 8 | imcheck  # check display region

(4x8) - r8g8b8o8
FIELD  MIN   MAX     3          2   BITS   1          0
-----  ---   ---    10987654 32109876 54321098 76543210
  r     30    48    ........ ........ ........ 00*1***0
  g     30    96    ........ ........ ........ 0******0
  b      0    30    ........ ........ ........ 000****0
  o      0     0    ........ ........ ........ 00000000
```

3.5. Imconst – Generate Constant Raster

Imconst creates a constant valued raster of arbitrary dimension and pixel specification. In many instances, the *-default* switch is used to borrow width and/or pixel specification from some "im" file, often subsequently involved in the overall synthesis. Values are specified using the format "-value aa:bb:cc..." where "aa..." are integers in the range $0..2^n-1$, and where n is the number of bits comprising the respective fields (the values pair left to right with the accompanying pixel field specification. This is one instance where the formats "r8g8b8" and "b8g8r8" are not interchangeable). When not specified, the *-value* flag defaults to "full on" (i.e., the maximum possible value for each field). This makes a logical '1' constant easily available for other tools, whereas a logical '0' can be always coded as such, independent of the field size.

Examples:

```
imconst -v 200:100:100 | imikw     # pink display (full screen)
imconst -d pos | imlop pos -op xor >neg.im          # flip bits
```

3.6. Imcrop – Rectangular Cropping

Imcrop performs cropping in the darkroom sense: the original data has margins removed from potentially all four sides and this inner window becomes the new output data. In particular, one cannot crop beyond the bounds of the original image. Although this tool is primarily used to crop in the visual sense, it is abstract in that cropping is done on a field-wide basis, and this indivisibility of pixels allows cropping of (for instance) image data containing Z-buffer depth information (or other pixel attributes) without danger of altering these components. As a corollary of this, the crop cannot (must not) perform any antialiasing along the clipping edges. Cropping is a very fast operation, as block copying can be performed. When no cropping dimensions are present, the rectangle defaults to the entire input file. In this last case *imcrop* resembles a *2-D* equivalent of the UNIX "cat" command.

Examples:

```
imcrop pix -x 72 -y 91 -w 1 -h 1 | imtabout   # pixel 72,91 is?
imcrop in -w 512 -h 512 >clip.im          # crop to screen rect
imcrop file | ...                      # resembles "cat f | ..."
```

3.7. Imcspace – Color Space Conversion

Imcspace is used to map three component color representations to and from RGB space. The color spaces supported are the NTSC defined YIQ space, the LUV and XYZ spaces defined by the Commission International L'Eclairage (CIE) and Smith's HSV space [Smit78]. Mapping is conformal in raster size and compression. The output pixels consist of three fields of eight bits, with the appropriate field names. Mapping RGB into a target space is specified with the *-space* {*name*} flag, with name $\in$ { *YIQ HSV LUV XYZ* }. The inverse mapping back into RGB space additionally requires the *-decode* flag. Mapping is done internally in floating point on the interval [0.0..1.0], with output data normalized to lie on this range. Thus, some colors in the YIQ gamut with negative coordinates (such as pure saturated yellow above 75% intensity) are clipped to lie in this range. This makes the tool useful in detecting or removing "hot" broadcast colors.

Example:

```
 # make "cool" for broadcast video
imcspace raw -space YIQ | imcspace -space YIQ -decode >tv
```

3.8. Imdegasr – Atari ST (Degas) Read

Imdegasr reads Atari ST display (Degas) files and produces standard *im* files for output. Trivial changes allow an Atari-based version to produce screen output directly. Width and height specifications are driven by the resolution of the input file and may not be altered. Pixel output is "r3g3b3" in accordance with the Atari hardware. The *-mapoff* flag may be used to return the unmapped pixel index, giving an output type of "n1", "n2" or "n4" as a function of the file resolution.

3.9. Imexpr – Pixelwise Infix Expression

Imexpr is a unary tool which provides pixel mapping based on a command-line specified infix expression. The tool is useful in performing general pixel-wise transformations for photometry corrections and in compositing mattes and screens. Output is of conforming size and of arbitrary pixel type. Each output pixel (specified by *-pixel*) has field values which are defined algebraically. These expressions may use any or all of the input fields as independent variables. The *-expression* flag accepts this pixel expression, with each field expression separated by ":" field delimiter. The expression uses conventional operator precedence with function and variable names represented by single characters in upper and lower case, respectively. Input and output data is normalized to lie on the range [0.0..1.0]. Evaluation is performed interpretively in floating point. The syntax is described fully in Table 11.

Examples:

```
imexpr newcolor -e ".3*r+.6*b+.1*b:#:#/3"  # old yellow tintype
imexpr bwimage -e "(n + 0.12) * 1.55"        # offset and scale
imexpr face -e "n*C(P*((X-.5)^2+(Y-.5)^2)^.5)" # round vignette
imexpr tone -p n1 -e "n*S(X*W)*C(Y*H)"    # sinusoidal halftone
imexpr bits12 -p n8 -e "L(255*n+1)/8"    # 8-bit log companding
```

3.10. Imextract – Multiple Merge/Extract

Imextract is a general tool which provides the only mechanism for the merging and extraction of pixel fields for dimensionally-conforming raster files. The *-field* switch specifies which fields are to be present in the output. Each output field must be unique within the collection of input files (other fields need not be). The output conforms dimensionally to the input files, with the fields taken (with no change of precision) from the proper input files. This generality allows one to permute the order of pixel fields as a unary operation on one file (not important for "im" tools, but a useful step when readying data for external uses), gather color separations into a composite file, or split a composite file into (a subset of) its components. The tool expects an arbitrary n files for input, each specified on the command line. The standard input may be included by the presence of a "-" flag on the command line.

CHARS	USE	DESCRIPTION
:	separator	between f1:f2:...:fn
-	unary minus	highest operator binding
^	exponentiation	(expr)^2 and (expr)^.5 fast
*,/	mult/div	$x/0$ is hardware dependent
+,-	plus/minus	lowest operator binding
( )	brackets	for expression nesting
a-z	variable name	must occur in input stream
0-9.	numeric const	formats: D D. D.F .F D.F
#	last field	use previous field's value
H	height	raster height (int)
W	width	raster width (int)
X	x position	current x [0.0..1.0]
Y	y position	current y [0.0..1.0]
P	π	useful for degree conversion
E	e	base of natural logarithm
I()	int	integer floor
S()	sin	(argument in radians)
C()	cosine	
T()	tangent	
A()	arctan	(result in radians)
L()	logarithm	base 2
J()	$J_0()$	Bessel function (1st kind)

Table 11 – Expression Syntax for *imexpr*

Examples:

```
 # merge rgb separations, omit Z data
imextract GZ.im BZ.im RZ.im -f RGB >RGB.im

 # extract overlay plane data
imikr -p r8g8b8o8 | imextract -f O >overlay.im
```

3.11. Imflatten – Convert Histogram to Map

Imflatten converts a histogram vector (represented by an $N \times 1$ raster image) into a "flat" map (table look-up) vector for performing histogram equalization [Hall71]. Under the mapping defined by this vector, all output pixels occur with equal likelihood, thus performing image enhancement. The tools *imhist* and *immap* proceed and follow the use of this tool in a typical pipeline.

Example:

```
 # generate equalization map, then remap input data
imhist milkdrop | imflatten |  immap milkdrop - >milkflat.im
```

3.12. Imfloyd1 – Convert Fields to One Bit (Error Diffusion)

Imfloyd1 is a level 1 algorithm which performs a scan line by scan line recoding of pixels into fields of one significant bit. A generalized Floyd algorithm with four-way forward error diffusion is used. The tool is useful (together with *imtobw*) in converting images for monochromatic devices, such as the Apple Macintosh. This halftoning can also be used on xerographic printing engines, but non-linearities in the transfer function of most hard copy engines imply that a number of adjusting steps are needed, making digital halftone dot techniques more attractive [Roet76]. The tool has been written to allow simple extensions to a more general setting, which allows for the arbitrary recoding of pixels. This data conversion is useful when converting twenty-four bit RGB into 8-bit RGB for display on many frame buffers. The tool also finds use in packing multiple images into one frame buffer to do color table animation of consecutive, repetitive frames in a fashion resembling old-fashioned cinematographs.

Examples:

```
imfloyd1 color24.im >color3image.im      # 8:1 pixel compression
imtobw color24.im | imfloyd1 >bin.im     # conv't color to 1 bit
```

3.13. Imhalftone – Digital Halftoner

Imhalftone provides a particularly efficient implementation of halftoning done by comparing a continuous tone input raster against a cyclic set of threshold weights, as defined by the auxiliary input. The program accepts no switches. The output is always one bit monochromatic "n1" data. Future versions might employ advanced thresholding techniques (such as adaptive minimization of threshold errors); this tool is the natural home for such improvements. At present, it models exactly the pipe-synthesized operation described in Chapter 2 and appearing in the first example.

Examples:

```
imtile dot -w 512 -h 512 |\              # halftoning using pipes
    imlop image - -op thresh | imtomask >half.im
imhalftone dot image >half.im              # identical operation

 # scale an image, print on Imagen
imsample in -s 5 | imhalftone dot | imcanw | lpr -Pimagen
```

3.14. Imheadoff – Strip Raster Header

Imheadoff is a switchless program which merely "eats" the header of an "im" file and then copies the raw raster data to the standard output. It is useful when creating raw raster data for subsequent delivery to external sites that do not support the "im" package. It can also be used to edit a header directly, by providing a "cut" operation. The accompanying "paste" is then done by file concatenation. This allow one to (for instance) rename pixel field names. The *-bytereverse* switch reverses the order of bits within each eight-bit byte on output. The *-invert* switch complements the output bits. These switches are often used together, as many binary hardcopy devices treat a '1' as a mark (black) and order the bits from left to right across the printed page.

Examples:

```
imheadoff oldfile | cat newhead - >newfile  # header transplant
imextract shipfile -f R | imheadoff >tapefl # ship red separate
```

3.15. Imheadon – Graft Raster Header

Imheadon is the inverse to *imheadoff*: it prefaces a raw raster with a text header specification taken from the command line (see also *imheadoff*). In particular, the command line may specify a *-default* switch to "borrow" specifications from an existing file. The program is used in conjunction with *imheadoff*, or when preparing external "raw" raster files for use within the "im" universe. The switches *-bytereverse* and *-invert* are identical in use to *imheadoff*. The *-skip* switch is used to ignore the first *n* bytes of the input file.

Examples:

```
imheadoff f4x4-r8 | imheadon -w 4 -h 4 -p n8 >a # turn r8 to n8

 # convert raw Versatec bits into an im file
imheadon v.bits -b -i -w 2112 -h 1700 -c 0 -p n1 >a
```

3.16. Imhist – Histogram Data

Imhist provides a histogram (occurrence tally) of pixels or fields. The output data is a vector in the form of a $W \times 1$ image file, with the width set to the maximum number of bins needed to encode the field of greatest bit significance. Histogramming of pixels with more than one field are tallied separately. Aggregate tallying (giving rise to very large vectors) can be specified with the *-aggregate* flag, which views all pixel fields as comprising one large integer.

3.17. Imikr – Ikonas (Adage) Display Read

Imikr is the general read utility and the unique agent for formatting Ikonas data in the "im" format. The offset and dimension switches specify the area to be extracted and the pixel switch (which defaults to r8g8b8) describes the layout of the pixel within the hardware device. The *-res* flag specifies that data is to be captured in a high-resolution (1024×1024) format; low resolution is chosen by default. Dimensional switches (*-xoffset*, *-yoffset*, *-width* and *-height*) operate as for *imcrop*. The *-shift* switch specifies the lowest order bit in the frame buffer from which data is taken; it defaults to zero. The *-lut* switch allows reading of the color LUTs, this is described in *imikw*. The program insures that the state (both pixels and registers) within the Ikonas is left exactly as is. Any operations intended on the pixels read in (such as hardware masking operations) are left to be done by subsequent software tools such as *imlop*. All Ikonas frame buffer read operations are embodied by this program only, so that the intricacies of its low-level software have a unique home.

Examples:

```
(ik0; imikr ) | (ik1; imikw)          # copy ik0 image to ik1
imikr -w 5 -h 5 -p g8 -s 8 >patch       # get 5x5 green plane
imikr -p o8 -s 24 | imtomask >ov        # bit mask of overlay
```

3.18. Imikw – Ikonas (Adage) Display Write

Imikw is the sole agent for writing pixels and color table entries to an Ikonas display, in a manner analogous to the companion *imikr*. As with its mate, an effort is made not to change any frame buffer state, except as needed. In particular, the program alters only pixels within the output rectangle specified, and then only those bits actually present in the pixel specification are altered, through use of the write mask.

The color tables and crossbar are necessarily altered to provide the correct RGB color space. Should a subset of "rgb" be present, only those separations present are updated on the display, and the color table and crossbar updated in a "partial" manner. The *-xoffset* and *-yoffset* switches may be used to offset the location of the image, which is imaged in its entirety (but clipped to the physical confines of the Ikonas memory). This size is ascertained by memory sizing code which runs during program initialization. Thus, it is impossible to draw (for instance) a "r8g8b8o8" image on a device with only twenty-four bits of pixel depth.

A number of hardware specific switches exist for this tool. The *-res* switch sets the hardware registers to high-resolution (internal sync) display mode, with an accompanying redefinition of the display and pixel size to the software. Future plans include a *-frame* switch useful in packing additional images in normally invisible portions of the display memory. The *-shift* switch allows one to slide the pixel data about within the frame buffer; pixel data is normally written into memory lsb justified, the shift switch inserts vacant (not written) bit positions to the right. When specified, the crossbar settings are shifted accordingly, so the visual effect is identical in either case. The *-zoom* switch uses the hardware zoom, pan and scroll registers to pixel-replicate the image to the largest size which still fills the display, scrolled so that (0,0) of the image corresponds to (0,0) of the Ikonas. This function may be invoked in addition to the *-res* switch, in which case algorithms as specified

in the Ikonas hardware manual (plus some additional experimental discoveries) are employed to generate the proper window.

The *-lut* switch specifies that the color look up tables (LUTs) are to be loaded instead of image memory. Here the *x*,*y* raster dimensions specify the table number and offset. The Ikonas provides four tables (only the first is normally used) limiting the input raster to a clipped size of 4×256. The *-xoffet* and *-yoffset* switches may be used to specify offsets in a manner analogous to image writing. The raster values specify gamma corrected LUT entries on the range [0..1]. A literal map copy may be made by specifying a raster with "r10g10b10" data and a gamma value of unity. Color lookup table loading may be disabled with the *-mapoff* switch; this also disables alterations to the crossbar switch. The *-gamma* switch expects a floating point value (default is 2.2) which gives the gamma of the monitor, used whenever color tables are loaded by the tool.

Examples:

```
imikw pix -x 100 -y 200 -r          # offset image, high res mode
imikw <bw8bit.im -z -s 24           # bw image in overlay, zoomed
imikw pseudomap -lut                # write pseudo-colors to LUTs
```

3.19. Imirw – Iris (Silicon Graphics Inc.) Display Write

Imirw works in a manner analogous to *imikw* to display images. For high resolution RGB output, the internal "RGBmode" is used to write images with a maximum size of 1024×768×24 bits, dependent on the number of available bit planes. Images requiring twelve or less bits use the "onemap" mode and color lookup tables for output. In this case, the indices and accompanying map locations are chosen to minimize collisions with commonly used system map locations. This mode is useful when simultaneous display of text is required or when less then twenty-four hardware planes are available. The *-lut* switch allows the direct loading for the color lookup tables. The Iris tables are loaded either as a 16×256 table ("multimap" mode) or a 1×4096 table ("onemap" mode), based on the width of the input raster. If run under the window manager the tool will prompt for the creation of an output window which is constrained to conform in width and height to the input.

3.20. Imkern – Convolution by Arbitrary Kernel

Imkern performs a convolution of the input file against the auxiliary input file (kernel). The output is of matching pixel attributes, but with spatial dimensions reduced by $W-1$ and $H-1$, where W and H are the respective width and height of the auxiliary kernel. Convolution is done in the integer domain, with kernel field components representing the integer multipliers for the input. An integer divide (with inherent truncation) normalizes the result by dividing the output pixel by the sum of the kernel weights. This step is omitted if the *-normaloff* switch is present, or if the sum of kernel weights is zero. Negative kernel weights are accommodated by specifying a positive *-offset* value which biases all kernel weights downwards by the specified amount. If the kernel contains pixels of exactly one field, this field is paired against each field of the input file during convolution. This is the common case, unless differential convolution by fields is required.

Examples:

```
imconst -w 2 -h 2 -def im | imkern im - >out  # low-pass filter

(echo "1 0 1"; echo "0 6 0"; echo "1 0 1") |\       # Laplacian
    imtabin -w 3 -h 3 -p n3 |\           # is a 3x3 filter, used
    imkern in - -off 1 -norm >lap.im     # to highlight gradients
```

3.21. Imlinear – Bi-linear Interpolation Filter

Imlinear provides bi-linear interpolation: it is most useful for scaling a raster to a larger size. Scaling is given by a *-scale* floating point value, with *-width* and *-height* values used to give exact output dimensions, in analogous fashion to *imsample*. Sampling is done by performing a bi-linear interpolation (weighted-averaging) of the four nearest neighbors to the output sample location. Thus, the tool is not atomic. The *-power* switch allows the weighing to shift from a linear ramp to a function of the fractional inter-pixel distance. This "peaks" the data, as $n \rightarrow \infty$ gives nearest neighbor sampling. Interpolation filtering breaks down for scaling below 0.5×, where *imkern* should be used for better results. The output sampling grid is designed so that scaling by 0.5× and 1× give results identical with box filters of size 2×2 and 1×1, respectively. Internal optimization reduces the number of multiplies to two per output pixel (field). The computation is carried out in either fixed or floating point, based on conditional compilation directives within the program. Fixed point operation can expect a two-fold increase in speed, at the cost of imprecision of the least significant bit of the output.

This algorithm is a "correct" first order approximation because the unity filter operation reconstructs the input data exactly and the algorithm is isotropic (axis independent). Whereas the "0th order" nearest neighbor method of *imsample* generates piecewise constant output when interpolating between pixels, this method's interpolation yields *2-D* piecewise linear output along the surface of a parabolic hyperboloid which interpolates the four corners. Consequently, the Fourier spectrum of this filter decays as $1/n^2$, not $1/n$, so high-frequency energy (noise) introduced by this filter is far less apparent [[Pant65]].

Examples:

```
 # anamorphic (xscale != yscale) scaling to fill screen
imlinear dog -w 512 -h 512 | imikw

 # fast shrink with some antialiasing, imbox is better
imlinear huge -w 10 -h 10 >thumb.im
```

3.22. Immap – Image Map (Color Table Look-Up)

Immap models hardware color map lookup operations. More generally, it allows for arbitrary input pixels to index into a table of output pixels. The mapping vector is passed as a $W \times 1$ image file where W can accommodate the largest value in any one field of the input data. Thus, the output conforms in spatial attributes to the input file and in pixel attributes to the secondary (auxiliary) input file. The present version accommodates input whose pixels contain only one field.

Example:

```
imrandom -w 256 -h 1 | immap pic - | imikw # pic in pseudocolor
```

3.23. Immargin – Add Margins (Borders)

Immargin augments an input file with a margin of arbitrary value specified by the *-value* switch and defaulting to full-on for each component. The margin (of some default size) is added to all four sides of the raster. The margin size is set by the *-margin* switch, which may be overridden or specified individually for each of the *-left*, *-right*, *-top* or *-bottom* switches. Additions of margins normally extends the width and height of the data to accommodate the larger size. The *-overlap* switch forces an output raster of conforming size, but at the expense of the overwriting existing values along the edges.

Example:

```
immargin picgrey8 -marg 6 -over |\     # generate white edging,
    immargin -marg 3 -v 0 -oper >pic8border # with black margin
```

3.24. Immedian3 – Median Filtering (3×3)

Immedian3 is a non-linear operator which computes the median value of pixels within a 3×3 box. Median filtering is most useful in reducing spike noise from input data, while preserving other high-frequency detail, such as edges. In this capacity it is more useful than low-pass (linear) filtering [[Huan81]]. The output file conforms in size and pixel attributes to the input. Median filtering within one pixel of any edge uses a reduced median window. Median finding is done on a field by field basis, so the operation preserves only pixel fields, not aggregate pixels. The program accepts no switches.

Example:

```
imrandom -p n3 -def pure |\          # conforming noise image,
    imtomask |\                  # mask with 1/8 holes missing,
    imaop pure -op mult |\           # pure image + dropouts
    immedian3 >fixup.im                     # fixed back up.
```


3.25. Imramp – Constant Ramp

Imramp produces a ramp (or "wedge") of increasing pixel values. Being a "source" tool, both spatial and pixel attributes must be specified. This version produces consecutive integers by pixel fields for all 2^n values across the width of the output scan line. Thus, *-width* must be set to 2^n, where n is the sum of the integers in the pixel component string. The switch *-height* must be set to one.

3.26. Imrandom – Random Numbers (Digital Noise)

Imrandom closely resembles *imconst*, except that random values are substituted for constant values throughout the grid. The *-value* switch specifies the upper range for the numbers, which are evenly distributed in the range [0..*value*]. If not present, *-value* defaults to 2^n-1 where n is the bit significance of each pixel field. Put another way, by default all bits of each field may be considered random, with an even chance of being either on or off. The program also allows the *-default* switch, useful in creating random grids with dimensions and pixel specifications conforming to some other source. *Note*: the program uses a C (UNIX system) pseudo-random number generator and incurs any related penalties for non-randomness. Independent runs of the program with identical parameters necessarily generate the same set of random values. A *-seed* switch can be used to reseed the generator for different values.

Examples:

```
imrandom -w 10 -h 10 -p r8g8b8 | imikw -z     # a digital quilt
imrandom -w 10 -h 10 -p r8g8b8 -seed 1 | imikw -z   # new quilt
```

3.27. Imrect – Find Bounding Box

Imrect makes a single scan through a raster file and identifies the width of the four margins, where a margin is "all background" in value. The output detail is written on the standard output in the form of a set of switches, for example: "-x 20 -y 15 -w 504 -h 460". This data can subsequently be used as cropping parameters for *imcrop*, by passing the values on the command line (using the shell back-quote directive). The program is useful both in trimming out unnecessary empty space from image repositories (but note: if stored as run-length encoded data, such lines typically compact quite well anyway) and in chopping and trimming data from external sources. Ideally, the program would output an "im" file, but this bumps the operation from level 0 to level 2, as potentially the entire file must be saved before a trimming decision can occur. A non-existent tool: *imtoptrim* would be a level 1 tool which would discard all initial blank scan lines while writing to the standard output directly. However, note that no other margin can be discarded in a level 1 setting without violating causality.

Examples:

```
 # crop a file to its minimum bounding box
imrect big >crop; imcrop big 'cat crop' >small.im

 # check two files to see if identical
imlop file1 file2 -op xor | imrect
-w 1 -h 512 -x 0 -y 0
```

3.28. Imrtr – Raster Technologies Read

Imrtr is a source tool which reads frame buffer images over a serial tty line. Models supported are Raster Technologies Model One series (1/25, 1/40 and 1/60). The device selected is specified by the *-device* flag, which defaults to the directory entry "/dev/RasterTech0". This entry is typically linked (aliased) to the physical device. The spatial data to be read is specified by *-xoffset*, *-yoffset*, *-width* and *-height* flags, using the left-handed coordinate system common to raster devices (*y* increases down the screen). Pixel data is specified by the *-pixel* flag, to a maximum of twenty-four bits.

Data is moved using either an internal eight or twenty-four bit protocol which uses run-length coding to minimize bytes transmitted. The output file may have an arbitrary *-compression* value, as the general *im* representation is independent of the file transfer protocol.

The source code for *imrtr* and *imrtw* provide a useful model for writing frame buffer device drivers. There is a clear line of separation between the raster manipulation functions and the device specifics.

Examples:

```
imrtr -p n8 -c 1 >bwimage.im              # grab full b/w screen
imrtr -d /dev/tty01 | imrtw -d /dev/tty02     # copy rt1 to rt2
```

3.29. Imrtw – Raster Technologies Write

Imrtw is a drain tool which writes to Raster Technologies Model One hardware. As a companion to *imrtr*, the *-device* flag is used analogously to choose the physical tty line used for serial data transfer. The location of the data written is specified by *-xoffset* and *-yoffset* flags, which define the distance from the screen origin to the image origin, using a left-handed coordinate system. By default, the frame buffer write mask register is set, allowing only the required pixel bit planes to change. A full cold-start may be initiated using the *-coldstart* flag. This step additionally clears the contents of the video memory, and places default color ramps in the color lookup tables. The color maps are not altered (this is an expensive operation), unless the *-mapset* flag is present, in which case the map is set to correspond to the current pixel type. When updated, the map uses the *-gamma* value to set monitor contrast. After power-up or cold-start, the hardware defaults to a linear map useful for displaying "r8g8b8" or "n8" pixel data.

Examples:

```
imconst -w 8 -h 8 | imrtw -x 252 -y 252  # central white target
imfloyd milkdrop | imrtw -m            # write one bit halftone
```

3.30. Imrotate – General Mirroring and Rotation

Imrotate performs mirroring by y, by x ("flipping") and transposition (by rows and columns), in any combination. Thus, it allows for arbitrary rotation and mirroring about the axes and either diagonal. Mirroring operations are performed "first" i.e., both "-mirror -trans" and "-trans -mirror" perform a counter-clockwise ninety degree rotation. This level two tool allocates storage to accommodate the entire raster, unless x-mirroring alone is required, in which case it operates on a scan line basis.

Examples:

```
imrotate face -t -m | imlop face -op av >inkblot.im # Rorschach

 # "flip" and zoom image for taking reversed slides
imrotate smallimage -f | imikw -z
```

3.31. Imsample – Nearest Neighbor Sampling (Scaling)

Imsample is used to scale an abstract image by using a nearest neighbor sampling function. In the case of resampling to a larger size, this necessarily implies some duplication of an input pixel. In fact, when integral scaling is called for, each pixel will be replicated the same integral number of times. The program is "level 0" in that no averaging of pixels is done, which is useful where pixel data includes non-intensity attributes, but is a "0th order" approach where general filtering is concerned. The program builds a DDA table of nearest neighbors in *x* and in *y*, skips to the next scan line where data is present, and then resamples along the output locations specified by this table. The output dimensions initially default to the input size; these values are both scaled uniformly by the *-scale* switch, which specifies a magnification value (*-scale* is a floating point value, and defaults to 1.0). If *-width* or *-height* switches are present, they override any scaling absolutely. Their use allows exact conformance to some output specification, and with it, the possibility for anamorphic (non-uniform across all dimensions) scaling. The *-xoffset* and *-yoffset* offset switches (which default to the origin) specify the location of the first output pixel. Their presence is helpful when scaling to a lower resolution. For instance, a 4:1 scaling to a smaller size samples every fourth neighbor along both axes, with the upper left corner of an imaginary 4×4 grid (tiled uniformly over the input) corresponding to the sampling point. Setting *x* and *y* to (3,3) resamples by taking the lower right (and not upper left) sample point of this grid.

Examples:

```
imikr | imsample -w 1024 -h 1024 >big.im     # bit double image
imsample in -x 1 -s 0.56 >pintsize.im     # scale down by ~ 2:1
```

3.32. Imshear – Scanline Shearing (Rotation Aid)

The tool *imshear* skews consecutive scan lines by an amount proportional to $\alpha \times row$, where α is specified by the *-shear* switch. The shearing implies pixel resampling (box filtering) along the x axis to do proper antialiasing. Thus, the tool is not atomic, but is a level one (constant scan line storage) operation. Shearing along the y axis can be achieved by surrounding this tool by rotation operations. Matrix shearing allows arbitrary 2×2 affine 2-D matrix operations of unit determinant to be performed on the raster. These include arbitrary and simultaneous shearing and rotation to the data. Scaling is excluded, as these violate the property of unit determinants common to all raster shearing. These operations are described in detail in [[Paet86]]. The present implementation uses fixed-point arithmetic for fast evaluation; conditional compilation code may be used for full floating point operation, at a speed penalty of around 2.5× on the VAX 8600 with FP acceleration hardware.

Example:

```
 # 45 degree rotation = xshear/yshear/xshear sequence
imshear faceup -s .4142 |\
    imrotate -m -t | imshear -s .7071 |\
    imrotate -f -t | imshear -s .4142 >faceat45.im
```

3.33. Imtabin – Text to Raster

Imtabin converts arrays of non-negative pixel (fields) into "im" format files. Although the range of use is wide, a typical use is in generating simple textures or regular arrays, which are then typically replicated to form arbitrarily large arrays. It also has some limited use as a companion with *imtabout*. In this mode, one can (conceivably) "dump" a raster file, edit it conventionally and finish the operation using *imtabin* to reconstruct the original data. Switches defining raster layout must be present on the command line (or the *-default* switch must be present) because raster height cannot be determined prior to reading all textual input. The tool is a level 0 operation. The format of the input data is "aa:bb:cc..." i.e., it is consistent with the value specification used for the *-value* switch and elsewhere. Input is from the standard input *only*.

Examples:

```
imtabin <edver.txt -d oldver.im >newver.im   # text edit image
imtabin -w 4 -h 4 -p n8 >n4x4.im   # enter a 4x4 h'tone matrix
100 200   0 255
255 0   200 100
150 255   0  50
0  200  100 255
^D
```


3.34. Imtabout – Raster to Text Dump

Imtabout dumps pixel data onto the standard output. The format is "height" lines of "pixel" length, with spaces between pixels and pixel fields set off by ':' characters. In particular, no header giving dimensions or pixel specification precedes the actual data. The *-align* switch is used to print each integer in a constant width field, thus maintaining textural alignment. The *-hex* flag specifies hexadecimal output. It is useful when creating input for raster systems which accept textual image data, such as PostScript. The output data is in the correct format to mate with the companion tool *imtabin*. On files with long scan lines, the line length might be too cumbersome for treatment with standard text editing tools. Concatenated with *imcrop*, the tool becomes a useful windowed-dump aid.

Examples:

```
imtabout img -h | tr -d ' 12' >ps  # packed hex for PostScript
imikr | imcrop -x 50 -y 49 -w 1 -h 1 | imtabout  # pixel 50,49?
20:40:20

imtabout colortexture                          # dump a small file
0:0:0 1:0:1 2:0:2 3:0:3
1:1:0 2:1:1 3:1:2 0:1:3
2:2:0 3:2:1 0:2:2 1:2:3
3:3:0 0:3:1 1:3:2 2:3:3
```

3.35. Imtile – Rectangular Tessellation

Imtile performs a rectilinear plane tessellation of the input data onto the output file. The output dimensions must be specified; pixel attributes are unchanged. This program is "level 2" in that the entire input file must be stored in main memory. Note that this is not the case where tiling along the horizontal direction only is needed; here one need only replicate on a scan line by scan line basis. In fact, *imtile* generates all "height" different full-length output scan lines internally, and then sends them to the output in a cyclic fashion. The benefit here is tremendous speed, as the program is often called to generate "stipple" grids. However, it means that in the worst case an $N \times 1$ array converted to an $N \times M$ array must allocate $N \times M$ storage, and not $N \times 1$ storage. The switches *-xoffset* and *-yoffset* may be used to offset the location of the first output pixel. These switch values must each lie in the range $[0..2^n-1]$.

Examples:

```
imtile smiley -w 512 -h 512 | imikw   # rubberstamp smiley face
imtabin <text4x4 | imtile -w 512 -h 512 >s      # h'tone screen
```

3.36. Imtobw – Convert RGB to BW

Imtobw converts RGB data of arbitrary pixel significance to monochromatic data using a table of weights (it is in fact a dot product which forms the Y component of the RCA-defined and NTSC-adopted YIQ encoding standard [[Limb77]]). The weights have a strong bias in the green, and follow the human eye's sensitivity closely. Thus, the conversion generates virtually no apparent shift in visual intensity of the image. The calculations are all done in floating point. *Note*: this program is identical in operation to

```
imcspace -space YIQ | imextract - -field Y ...
```

A more general program would treat each input pixel as a vector (with fields forming the elements) of size N, which when postmultiplied by an $N \times M$ matrix, would output a pixel of M fields.

Examples:

```
imikr | imtobw >tmp.im                  # b/w version of display
imtobw colpix | imfloyd1 | imversw      # convert, then halftone
```

3.37. Imtocolor – Convert to Twenty-Four Bit RGB

Imtocolor is particularly useful when merging a number of pictures of various bit significance into a master image for viewing on the Ikonas frame buffer. A more general tool extends *imlop* to deal with images of conforming field names but with varying degrees of significance; for the moment this tool is provided to provide general composition capability on the Ikonas screen. In particular, the tool converts any field named "n" or "y" (else the very first field present) into an "r8g8b8" pixel by a simple multiplication and replication of field values. The program should "no-op" when RGB data is passed (or convert field by field into eight-bit significance), but this has not been done, as the projected lifetime of this tool is not long.

Examples:

```
imtobw colpix | imtocolor | imikw       # write with b/w version
imfloyd1 n8file | imtocolor | imikw     # binary file as r8g8b8
```

3.38. Imtofont – Convert to Berkeley Font

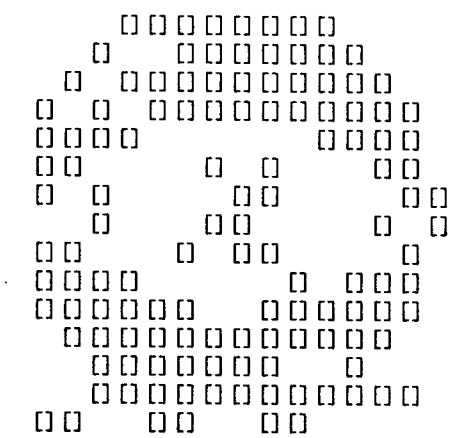
Imtofont converts arbitrary files (typically bi-level ones) into font rasters for conversion into "Berkeley" fonts. From here, manual steps exist to integrate the font into libraries for printing on the Imagen laser printer. In particular, the tool makes possible the creation of custom font logos, taken from some "im" manipulated source. At present, the tool provides merely a text dump of the raster given (with non-background data generating the text "[]", analogous to output from the Berkeley font tool *vfontinfo*). The input raster is typically a bi-level file, created through some halftoning technique. The character raster output is prefaced with the necessary text to run "devnew" and related font software to build up a single character glyph. A more ambitious tool would update the glyph(s) directly within the font file, without need for an intermediate textual representation. The program takes no switches.

Example:

```
imsample eyeball -s 0.25 | imtofont # map mazewar eye into font

Font X.10, raster size 2310, max width 16,max height 16,xtend 0

 ASCII      offset     size  left    gwidth  top     gheight width
 1 ^A         0         0      0      16      16       16     16
```



(Original)

3.39. Imtomask – Record Non-Zero Pixels

Imtomask generates a bi-level "n1" image whose output is '1' everywhere a non-background color appears in the input, and '0' otherwise. Our choice for background is for a few reasons. It is a simple extension of the common practice that the integers zero/non-zero are used to represent the booleans false/true, in languages such as LISP and C. These values are also both unique and are present in any definition of pixel fields (forming a useful additive identity for arithmetic pixel operations). Finally, our laboratory software already supports the concept of "under" and "over" when rewriting images to a display; these operations are in fact binary operand filters which choose the pixels from the proper input stream, based on whether that input is "background" or not. Thus, we extend the notion of background in a fashion which embraces current usage. This operation is different from the "imlop -op thresh" operation against a threshold of zero, here non-background values are represented as '1' in the format "n1", and with more than one field present, *all* fields must be zero (logical AND function). In contrast, *imlop* thresholds multiple fields separably, with "on" foreground values replaced with the maximum intensity possible for each field.

The intended use of masks is as a time saver and conceptual aid when doing multiple overlay composition, in the manner that a darkroom can do "dodging and burning", or as a video studio does "chroma keying". A mask represents some image area which is to be (in)active, and masks can be complemented, combined (or-ed) and the like, and then ultimately used to mask detail into or out of a picture. Whereas a complete picture may serve as a mask (it also has foreground

and background), they incur a storage penalty, particularly where many masking operations are performed.

Example:

```
imconst -d pic -v 40:20:10 |\          # make chroma-key mask,
    imlop pic -op xnor | imtomask # with cutout on 40:20:10 key
```

3.40. Imunmap – Convert Pixels to Indices

Imunmap is a binary operation which reverses the operation of *immap*. Given an input file and auxiliary map file of conforming pixel type, each input pixel is replaced by the index (x location) of the matching pixel in the map file. If a matching value cannot be found, the value zero is returned. No provision for near matching exists. The program accepts no switches. The program is most often used with immakemap for recovering color map and pixel data on images with a small number of unique pixel values.

Example:

```
imunmap col24 ctab16 >colorbit4.im  # unmap to 16 unique colors
```

3.41. Imversw – Versatec Output

Imversw copies "im" files directly to the Versatec printer, with I/O calls used to place the device in "plot" mode, starting at the top of the next page. At present, pages are masked to a size 1700×2112, and printing is done in landscape mode; thus, images can never cross the perforations. A black mark is printed for each background (zero) pixel field, the field used is the first named "n" or "y" (else the very first field present); in practice the images are "n1" data from tools such as *imtomask* or *imfloyd1*. The *-xoffset* of *-yoffset* switches are used to offset the location of the data; this is typically used to center the image or set margins beyond the pinch rollers on the paper feed. The offset values are in machine units, which for most Versatec class devices is two hundred dots per inch.

A number of additional switches are available. Because most printing engines interpret '1' to mean "print black", a *-invert* switch has been provided to allow this interpretation, otherwise (default) images print in the correct sense. The invert flag inverts only the printing bits; the non-printing areas of the page remain white. The *-spool* switch redirects the output to the standard output for subsequent queuing by the *lpr* line print spooler, using its *-v* option.

Example:

```
imfloyd1 art | imverw -s | lpr -Pver -v # h'tone convert, queue
```


3.42. Imxw – X Window Write

Imxw is an experimental tool for converting data into X window output specification, common on DEC/GPXII and SUN workstations. Full-color or monochromatic images of arbitrary precision are supported. No offset parameters are provided, as the tool spawns a window with size matching the image, possibly scaled (using bit replication) by an integer specified with the *-zoom* switch. The window has a black border of size defined by *-margin*. The *-geometry* and *-display* switches are X window parameters which allow for more general specification of the window or output device. For display, the tool allocates up to *-maxvals* lookup table entries, divides them (as a Cartesian product) across the RGB color space and uses a simplified Floyd error diffusion algorithm to draw the images. When the partitioning of the three discrete indices yields different amounts of precision, the intervals are assigned in GRB order, thus matching the eye's spectral sensitivity. This is the case when the default *-maxval* of two-hundred ten indices is used, as $210_{rgb}=6_r\times7_g\times5_b$. Should the image show a strong color bias, the *-bluebias* or *-redbias* flag may be used to choose index assignment in BGR or RGB order. The *-lut* flag may be used for direct loading of the 1×256 LUT. The monitor gamma may be set using the *-gamma* flag. A mean default value of 2.4 was chosen based on empirical observation.

References

[Adob85] Adobe Systems Inc., *PostScript Language Reference Manual* Addison-Wesley, Reading, Massachusetts, 1985.

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

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

[Cohe81] Cohen, D. "On Holy Wars and a Plea for Peace" *IEEE Computer* **14(10)** October 1981.

[Fium83] Fiume, E., Fournier, A. "A Parallel Scan Conversion Algorithm with Anti-Aliasing for a General Purpose Ultracomputer" *ACM Computer Graphics* (SIGGRAPH '83) **17(3)** July 1983, pp 145-147.

[Floy75] Floyd, R. W., Steinberg, L. "An Adaptive Algorithm for Spatial Gray Scale" *Society for Information Displays* International Symposium 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.

[Hall71] Hall, E. L., *et al*, "A Survey of Preprocessing and Feature Extraction Techniques for Radiographic Images" *IEEE Trans. Comp.* **20(9)** Sept 1971 pp 1033.

[Have82] Havens, W. "Standard Image Files and Bit Stream I/O" **TR-82-10** Department of Computer Science, University of British Columbia, Vancouver, Canada.

[Higg86] Higgins, T. M., Booth, K. S. "A Cel-Based Model for Paint Systems" *Proceedings, Graphics Interface '86*, Vancouver, pp 82-90.

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

[Huan81] Huang, T. S. (ed) "Two-Dimensional Digital Signal Processing II, Transforms and Median Filters" *Topics in Applied Physics* **v(43)** Springer-Verlag, Berlin 1981, pp 161.

[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* sec **6.7** (v.1), Springer-Verlag 1979, pp 234-239.

[Land83] Landy, M. S., et al "HIPS: A Unix-Based Image Processing System" *Computer Vision, Graphics, and Image Processing* **25** 1984, pp 331-347.

[Limb77] Limb, J. O., Rubinstein, C. D., Thompson, J. E. "Digital Coding of Color Video Signals – a Review" *IEEE Trans. Communication* **25(11)** November 1977, pp 1349-1385.

[Newm79] Newman, W. M., Sproull, R. F. *Principles of Interactive Computer Graphics* (2nd ed) McGraw-Hill, New York 1979.

[Oust81] Ousterhout, J. "Caesar: An Interactive Editor for VLSI Layouts" *VLSI Design* **2(11)** 4Q 1981.

[Patt85] Patterson, D. A. "Reduced Instruction Set Computers", *Communications of the ACM*, **28(1)** January 1985, pp 8-21.

[Paet86] Paeth, A. W., Booth, K. S. "Design and Experience with a Generalized Raster Toolkit" *Proceedings, Graphics Interface '86*, Vancouver, pp 91-97.

[Pant65] Panter, P. F. *Modulation, Noise, and Spectral Analysis* McGraw-Hill, New York 1965.

[Port83] Porter, T. "The Format of Stored Pictures" Technical Report #46, Computer Division, Lucasfilm Ltd. 1983.

[Port84] Porter, T., Duff, T. "Compositing Digital Images" *ACM Computer Graphics* (SIGGRAPH '84) **18(3)** July 1984, pp 253-259.

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

[Roet76] Roetling, P. G. "Halftone Method with edge enhancement and Moiré Suppression" *Jour. Opt. Soc. Amer.* **66(10)** October 1976, pp 985-989.

[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.

[Stev82] Stevens, W. R., Hunt, B. R. "Software Pipelines in Image Processing" *Computer Graphics and Image Processing* **20** 1982, pp 90-95.

[Tani80] Tanimoto, S., Klinger, A. *Structured Computer Vision: Machine Perception through Hierarchical Computation Structures*, Academic Press, New York 1980.