Chapter 22 – GUI Integration and Virtual Machine Constructions for Image Processing: Phenomenological and Database-Engineering insights into Computer Vision

Nathaniel Christen[1] Amy Neustein[2]

[1]Lead Software Architect, Linguistic Technology Systems, Fort Lee, New Jersey, USA
[2]Founder and CEO, Linguistic Technology Systems, Fort Lee, New Jersey, USA

Abstract

This chapter will be focused on image-processing and Computer Vision. We will consider requirements for a hypothetical Virtual Machine whose intent is consolidation of access to image-analysis functionality by exposing sets of Computer Vision algorithms through a common interface. We anticipate the image-related discussion by reviewing potential Virtual Machine features that are not unique to image-processing but may be especially relevant to that domain, such as recognizing different varieties of number-pairs as distinct built-in types (to represent, for instance, image locations/coordinates and/or geometric lengths, or pairs thereof). We also consider functionality that becomes consequential insofar as GUI front-ends are engineered alongside image-processing technology — either through user-guided algorithms (such as interactive segmentation) or through interactive visualization wherein users review workflow architectures, intermediate results, color histograms (and other special-purpose GUI components), or comparisons between distinct processing algorithms; and similar tools helping users select and fine-tune image-analysis techniques for specific image series. By way of providing an example or case-study for algorithms that may be integrated with Virtual Machines or application GUIs, we summarize some facets of a novel database-related image format and explain the mathematical background of its color models and in-memory layout for image data.

Keywords: Computer Vision, Image Processing, Image Formats, Database Engineering, Type Theory

1 Introduction

This chapter will use image-processing as a domain to examine issues related to Virtual Machines (VMs). We will use image-processing algorithms as a case study in the sorts of workflow modeling and quantitative operations which might reasonably be incorporated in VM designs, albeit a VM specialized for a specific computational domain. For the sake of discussion, this chapter will base some examples on a novel, rather idiosyncratic image format that we we have called "XCSD" (the acronym stands for "Extensible Channel System" and "Subdivision Indexing," to be clarified later). This novel format is optimized for certain image-processing operations related to approximate segmentation and color-based (more than shape-based) background/foreground separations. We will orient the exposition around hypothetical VMs for which XCSD supplies the "native" image format, and which might be used with image databases (e.g., for query evaluation) storing content in this format.

In the case of image-processing, there are literally hundreds of algorithms that might be run to transform, or extract information from, a 2D image (not to mention image-series, videos, or 3D point clouds), so mapping each algorithm to kernel VM instructions is impractical. Obviously, even an image-oriented VM will have to provide some foreign-function interface where various algorithms could be registered and accessible through VM code; the VM runtime could accordingly be provided with foreign-function pointers through the application which hosts and initializes the VM itself. On the other hand, there might be some operations related to image-processing which are so fundamental that they should be recognized by a domain-specific VM automatically, without needing an extra step of loading "foreign" functions (providing the relevant capabilities).

Preliminary to discussing algorithms directly related to images, we will review several themes arising in that context but applicable more broadly, within the overarching subject-matter of reasonable scope and operation-sets for (relatively high-level) VMs. This initial discussion will highlight type-theoretic issues and primitive mathematical functionality that go beyond the ubiquitous integers, floats, and fundamental arithmetic capabilities that one would expect from any VM (however broad or narrow in scope, and high-level or low-level in design). Whether or not these specific types and operations are appropriate for a specific VM, given its goals and motivations, these examples will hopefully suggest how the question of what are legitimately "fundamental" or "low-level" types and calculations is open-ended.

Numerous constructions usually associated with high-level languages that compile to VMs or intermediate representations (perhaps only in a temporary sense in the course of being compiled to actual machine code) may prove to be worthy of direct support and implementation at the VM level — aside simply from convenience (insofar as certain high-level operations do not need to be repeatedly translated down to multiple VM instructions) the specific conditions and rationales for the relevant high-level types can intersect with VM opsets in ways that justify accommodating them directly. Again, this chapter will illustrate such points with concrete examples. Subsequently, we will situate the types and operations reviewed in the preliminary discussion in the context of image-processing, so that first analysis will extend into Computer Vision as a case study for (what one might call) "domain-specific" VM design.

2 Type-Theoretic Constructions at the Virtual Machine Level

The types which lay at the foundation of almost any computer system's type hierarchy are integer and floating point values with different byte

lengths — often 1, 2, 4, and 8 byte integers (distinguishing signed and unsigned types) and 4 or 8 byte floating-point numbers (decimal approximations). It is less common for relatively low-level environments, such as conventional Virtual Machines, to model *pairs* of integers or decimals as "base" types, assuming that such values would more likely be implemented as types in a high-level language (e.g., **std::pair** in C++). However, number-pairs are such an essential component of image-processing operations that a VM specifically targeting this domain would have reason to recognize pairs as equally fundamental to single values; since any pixel or location in an image needs two independent numbers to be identified, and since image-locations are the building blocks around which Computer Vision algorithms are constructed, such two-number pairs are if anything more "foundational" than single values.

For similar reasons, one could equally argue that several *different* number-pair types are foundational. One consideration for image processing is that different Computer Vision libraries utilize different conventions about how positions within an image are to be notated. A detail which varies across platforms concerns where location-axes have their origin point: one could have the top-left corner correspond to the point (0, 0) — with axes increasingly down and to the right — or alternatively the origin at the bottom-right (actually, any of the corners) or even the image center.

Another question is which of the two axis-values — the horizontal or vertical coordinates — are notated first in a number-pair. These are not particularly complex details, of course, but care must be taken to ensure that analytic procedures are called with the proper location-encoding; i.e., that there is alignment between libraries' and calling procedures with respect to location-coordinate conventions.

Ensuring such alignment can be promoted via strong-typing: assuming that number-pair types are further narrowed to reflect coordinate conventions, only locations encoded according to the target expectations can be passed to the corresponding procedures. Such policies might include how coordinates are named: e.g., one common vocabulary for pairs, as in tables or spreadsheets, are to identify horizontal "lines" as *rows,* and vertical as *columns.* Moreover, row/column pairs usually notate the *row* coordinate first. Therefore, constraining a pair to be specifically *row/column* documents that the expected interpretation of the first coordinate is vertical (i.e., a vertical distance from the origin) and the second is horizontal. Conversely, a different popular convention (reflecting mathematical norms) is to label horizontal values as **x** and vertical as **y**. In this case, normally the **x** value comes *first*; which illustrates possible confusion, because a number-pair would be interpreted differently depending on whether it is understood to be row/column (vertical then horizontal) or x/y (horizontal then vertical).

To add further confusion, some procedures accept inputs in terms of *lengths*, e.g., width and height, where similar procedures would accept location-coordinates. So, without full contextual information, it is ambiguous whether a pair designates a coordinate or two length measures. Consider a procedure to calculate the average color or dominant color within some region-bound, where the inputs must define a rectangle spanning the focus of the computation. That rectangle might be described via a pair of locations — top-left and bottom-right corners, for instance — or alternatively via *one* coordinate-pair and one length-pair, e.g., top-left *corner* plus *width* and *height.* In other words, the second number-pair might be a *location* (x/y or row/column) or might be *dimensions* (width/height). Confusion can be avoided by having a width/height type type distinct from location-types, analogous to distinguishing row/column from x/y location coordinates.

At the same time, note also that some procedures might take number-pairs which do not model geometric quantities such as position or length, where terms like "width" or "column" could be misleading — in the generic C++ **std::pair**, for example, the first value is called "first" and the second "second." For such "non-geometric" pairs one might indeed prefer still another pair-type with coordinates labeled as first/second, say. Indeed, code for the XCSD format (for example) supports numerous variations on pair-types, including options based on start/end, top/bottom, left/right, horizontal/vertical, and above/below (e.g., if a pair's C++ type is, say, **tl2** — meaning top/left, two-byte values — the first coordinate would be accessed by a **.top** member and the second by **.left**).

Even were a virtual machine to natively model number-pairs, theorists preferring relatively minimalist VM design might think to provide one such pair-type (or perhaps several distinguished only by byte-length), anticipating that higher-level distinctions between conceptually distinct but structurally isomorphic types (like row/column, x/y, width/height, and first/second) would be made by source code, not VM byte-code. Ensuring that strong pair-types are never mismatched would then be the responsibility of the compiler emitting VM code, not the VM runtime. Consider, however, that pair-labeling could reasonably be an issue within special contexts, such as query-evaluation: an image-database might, for example, allow for queries which invoke the rectangle-focused procedure mentioned above (along the lines of "find images whose central third rectangle color-averages" to nearly some given shade). Assuming type-level distinctions such as x/y versus width/height are recognized by the query *language*, then query *evaluators* would of course at some stage ensure that an x/y coordinate is not passed to a rectangle constructor which expects width/height dimensions, say.

Moreover, the "compilers" that would bridge *query* code to VM instructions might be significantly different than those working on code from other genres, such as scripts (with notions of types, scopes, and variables closer in spirit to compiled programming languages). These circumstances could argue for stronger type distinctions being recognized even at the VM level, at least to some degree of detail, to help guarantee that type-enforced conventions are applied in similar ways across all environments that use the VM, notwithstanding their distinct compilation models (database queries compared with scripts, for instance).

From this perspective, then, assume that a VM is indeed designed to model *several* number-pair types. This decision immediately has implications for the *operations* supported through the VM, because some operations seem intrinsic to managing paired quantities; for instance, transpose (switching the first and second values) and arithmetic operations (which might involve a single value — altering both numbers by the same amount — or a second number-pair, applying the relevant operation between the first and seconds pairwise). Here is a more complete list of some operations which may be used often when working with number-pairs, and would be candidates for native encoding in a VM context:

(strictly) Ascending, descending, spaceship As boolean operators, return true if the second value is larger (respectively, smaller) than (or equal to, for the non-"strictly" options) the first. The "spaceship" version (borrowing the terms from many high-level programming languages) would return 1, 0 or -1 depending on whether the second

value is greater, equal, or less than the first.

Zeros mask, nonzeros mask, nonnegative mask, spaceship mask Return pairs whose components are one or zero depending on whether the two input pair-values are zero (respectively, are not zero; and, are at least zero). Similarly, "spaceship mask" is a plausible name for a function that applies a unary spaceship operator to each coordinate in a pair: yielding a new pair, each member either 1, -1, or 0, depending on whether the respective number in the original pair is positive, negative, or zero.

Area, ratio Product of the two numbers (or perhaps absolute value of the product); respectively, ratio of the second to the first (or perhaps a version which transposes descending pairs before taking the ratio, so the result is always at least one, or vice-versa, so the result is at *most* one).

Binary merge Combines the two numbers into a single value where the former numbers occupying different bit-positions (this operation could have several variants wherein one of the two inputs is bit-shifted by different amounts).

Maximum, minimum, difference, positive-integer difference, gap Greater or lesser of the two numbers; or, return the second subtracted from the first (the "positive integer" variant would result always in a non-negative output, mapping negative differences to zero), or "gap" between the two numbers as an absolute value (i.e., maximum minus minimum).

Duplicate, "end at" Given one number, constructs a pair by setting the second value to have a desired gap greater than the first (or, allowing this parameter to default to zero, form a pair with two equal numbers); alternatively, modify the second number in an already-existing pair to conform to such a gap.

Linear intersections Given two pairs, form a new pair which takes one value from each pair, analogous to the −| or |− operators in diagram languages such as **tikz** (in effect, define a third point as the middle vertex of a right-triangle whose hypotenuse is the given two points, thus the intersection of two lines, one horizontal and one vertical, including the two input points; there can be two versions for such a function, corresponding to the two extra corners of a rectangle if the given two points are one diagonal, or equivalently which input has the horizontal line and which the vertical). As a variant form, take one pair and then two other pairs (defining a line) and calculate the nearest distance of the former to the latter.

As arithmetic base Treat the two numbers as components of a two-digit number in some base (provided as a parameter): multiply the first number by that input, and add the second. This operation can be extended to value-triples recognized as distinct data types analogous to pairs, where the first number would be multiplied by the square of the input.

Manhattan, Chebychev, and other distances For cases where pairs represent 2D locations on an integer grid, there are several possible metrics that could be used to represent lengths (via *integers*) between grid-points — of course, one could compute real-valued Euclidean distances (approximated by floating-point numbers) but we'll focus instead on formulae expressed solely through integers (i.e., distances use the same type as point-components themselves). Familiar choices are the "Manhattan" metric, which calculates distances by counting orthogonal steps leading from a start to an end point), or the "Chebychev" metric, which counts the minimum number of orthogonal *or* diagonol steps, along grid intersections (a "grid-diagonal" would be the shortest diagonal between two grid-points, in effect two single orthogonal steps at right angles). Supposing $\{x, y\}$ and $\{a, b\}$ are two locations, the Manhattan and Chebychev equations would be $|x - a| + |y - b|$ and $\texttt{max}(|x - a|, |y - b|)$, respectively. Also, the XCSD format is built around a combination of these two distance measures (analyzed in the next section). One benefit of a Manhattan/Chebychev "hybrid" (at least in the form discussed below) is an almost-Euclidean precision with respect to ordering distances, while retaining only integer-based expressions: specifically, two location-pairs on an integer grid will only have the same Manhattan/Chebychev hybrid distance if they have the same Euclidean distance (and vice-versa). In any case, each of these (Manhattan, Chebychev, and the hybrid) are comparison operations that would be appropriate for any image-processing or geometry library.

Integer-based line operations The previous item reviewed several distance metrics appropriate for discrete-geometric systems. We can similarly incorporate other computations appropriate for geometric work restricted to integer-based mathematics. For example, consider the problem of calculations pertaining to the Bresenham's line between to points (i.e., the points drawn for graphics displays approximating a smooth Euclidean line, accommodating the restriction to a rectangular pixel-grid) [38], [5]. Given two end-points and a single intermediate coordinate, one might want to calculate the intersection of the formers' Bresenham's line with the horizontal or vertical axis-translation through the latter, or then go on to select some set of points adjacent to that intersection on either side (this last step would need information about the line specifically; it could not be ascertained indirectly via Euclidean calculations).[1]

Floor, ceiling, fit in range Modifies a pair's components so they are at least (respectively, at most) some value. Similarly, given two pairs, modify the first pair (as needed) to satisfy range-restrictions expressed by the second pair. For example, alter each number (of the first pair) so that it is no less than the first coordinate of the second pair, and no greater than its second coordinate.

Binary insert Treating the first number as a bitset, encode the second number through a designated bit-sequence within it (as a unary operator, simply shift the second number left by some count of bits, applying binary or of the result to the first; as a two-argument function, the second input could specify an *end* bit, so in effect the second number would be truncated of leftmost bits as needed and then copied into the first using a bitmask-guarded "**or**" operation which only alters bits in the specified range).

Most of the aforementioned operations are used in the demo code associated with this chapter (largely for calculations related to the XCSD image format), so this code can furnish examples of these operations employed in concrete situations.

One benefit of providing a relatively thorough suite of operators defined for number-pairs is encouraging programmers to think of algorithms in terms of operation-sequences on number pairs as integral units, not as sequences applied to the two pair-elements in isolation. Usually pairs designate some logically connected entity (e.g., a geometric point) and operations have an interpretation where it is conceptually accurate to consider the pair as one value in a two-coordinate space, rather than two separate numbers. As an example, XCSD has a function called "quadrant_code_against" which is defined (within a C++ macro) as a series of pair-operators:

[1] An example of where this sort of problem might arise would be implementing mouseover effects where we want to highlight some region of a Bresenham's line as a visual cue for cursor-location or the image-point where some action would take effect.

```
u1 quadrant_code_against(const ty##size##s& rhs) const
{
 return (*this - rhs).spaceship_mask()
  .plus((*this - rhs).zeros_mask())
  .floor(0).times({2, 1}).inner_sum();
}
```

That code may not be more readable than alternatives which work with each coordinate separately, but it fits within a paradigm where we are encouraged to think of pairs as self-contained mathematical entities whenever possible.

Although the names chosen for some of these operations make more sense in some types' contexts than other ("area" as a label for the inner-product absolute value fits width/height better than row/column, for example), it seems reasonable to provide operators for all versions of structurally similar types absent compelling reasons to make specific exceptions. Moreover, distinctions such as row/column versus width/height (and x/y, first/second, top/left, etc.) are orthogonal to signed/unsigned, integer/float, and byte lengths for the components. Accordingly, we can derive a fairly large number of "primitive" pair-types, reflecting different combinations of coordinate names, signed/unsigned, 1/2/4/8 bytes, and floating-point-based types (albeit some operations applicable to integers, such as ones based on bitsets or binary encodings, would not carry over).

On the principle that most operations available for one pair-type should be available for others, this results in quite a few duplicate operations differing only by input types. Assuming that high-level coding techniques such as templates are not available for VM implementations, such a scenario seems to demand brute-force coding of multiple function-versions (perhaps aided by code generators). It is not difficult to maintain a function-pointer table with hundreds of entries if needed, so supporting this diversity of "primitive" operations does not impose a major overhead for a VM runtime. However, once code-generators (and perhaps more complex grammars) are involved, the development environment for *implementing* a VM runtime would become relatively more complex, and would need to be set up properly.

Since we are not constrained by existing VM or intermediate representations, there's room for creativity in how opsets are designed, in terms of their textual labels (or even their numeric opcodes) from the point of view of people reading "virtual assembly" code for the VM, as well as runtime engineering (opcodes might have bit-flags carrying information about their respective operators, for example), with the idea of organizing groups of operations differing only by (e.g.) input/return types. Likewise, a well-designed VM infrastructure can allow designers of high-level languages targeting the VM some creativity as well. The pair-operator code for XCSD mentioned above (which expands via macros to C++, not a VM, but could migrate to other contexts) is constrained by C++ syntax and operator-overloading, but other VM-specific languages would be free to, for instance, create new binary operators for pairs. We alluded above to one example: the **tikz** −| and |− could be used as syntactic atoms in a high-level language, compiling to line-intersection functions (analogous to the <=> spaceship recently being added to the C++ operator set, in current specifications for that language). Similar comments could be made for other operators which seem common enough that they might warrant operator-like syntax (for readability) rather than function-call notation — examples might be min/max, absolute value (applying the "**abs**" operator to both components), floor/ceiling, ranges, and so forth.

Also, a comparable custom-syntax scenario might apply to conventional arithmetic (and boolean/binary) operators in contexts where it is necessary to distinguish an operation between a *pair* and a *single* value (e.g., multiplying both coordinates by the same scale), or conversely another pair (e.g., vector multiplication[2]), or (alternatively again) *internally* within one pair. For instance, there are three important versions of, say, addition in a number-pair context: given $\{x, y\}$ there is $\{x + s, y + s\}$ (for some scalar s), or $\{x + a, y + b\}$ for pair $\{a, b\}$, or $x + y$ (the "inner" sum, a scalar result).

These alternatives would similarly exist for almost all "primitive" numeric operators for most coding languages, which would include arithmetic functions and also boolean calculations both in the true-false sense (e.g., treating positive or non-zero values as "**true**" and zeros as "**false**") and bitwise manipulation, plus binary operations such as left- and right-shift. A high-level language might notate these variations via compound tokens combining one part or character representing the operation itself and another character (or characters) indicating a specific variant (as a hypothetical example, a bare + might represent addition *between* pairs, while .+ represents "inner sum" and +. encodes addition with a duplicated scalar).

Apart from overloading functions for many structurally similar types, there are further dimensions to the overlap between VM operations, types, and code-management, some of which are indirectly posed by the previous examples. Consider the process of casting a pair from signed to unsigned. Plausibly, this might be achieved in several ways: first, take absolute values; second, map negatives to zero (essentially using a "floor" operator with parameter zero); or, third, simply reinterpret any negative value as the binary encoding of a positive value (so -1, say, would become the maximum integer of the corresponding unsigned type). Each of these could potentially be expressed as operators between pairs as well as type-cast logic; as such, one consideration when formulating a VM's underlying type system is to define operators which provide the calculations for type-casts (in case someone wants to achieve type-cast-like effects via a function-call, or perhaps to enable scenarios where the VM could be configured to perform type casts in different ways by specifying a desired operator).

More generally, it's worth noting how type-casts intersect with operator-application: consider an "inner difference" function which subtracts the second pair-value from a first. Assuming we sustain the principle that unsigned pairs should by default perform operations yielding other unsigned pairs/values, we might want the *non-negative* version of inner-difference, which in turn implies casting a signed result to unsigned. This is analogous to a "nonnegative-inner-difference" procedure which applies subtraction but with a floor of zero — however, such a variation might *also* be useful in a signed (including a floating-point) environment, or as the nullary version of a unary operator which takes an inner-difference raised (as needed) to a floor which might be something *other* than zero, passed as a parameter.

2.1 Issues with overflow/underflow and loop termination

Similar ideas apply in the context of addition with possible overflow: should be consider addition involving one-byte integers to be capped at 255, or to "wrap around" past zero? The latter option does not make much sense in the case of colors, say (it would almost certainly be an error if an algorithm considered "pure black" zero to be one unit greater than "pure white" 255), and moreover could be a notorious source of infinite-recursion bugs (cf. tests for <=255, which might never revert to false, depending on the type involved). Types related to colors (in conventional one-byte-per-component encodings) are good examples

[2] In the sense of Hadamard product; cf. Julia's **.*** operator.

of types for which basic arithmetic operations should almost always be guarded against overflow (addition maxed at 255 without circling around to zero, and subtraction floored at zero without negative-overflow back to 255). Conversely, in some contexts integer-maximum wraparound effects are exploited deliberately; analogously, casting an integer to a single byte (typically in the context of merging multiple values via bitmasks) performs a function analogous to zeroing out bits with index 9 and greater.

The overarching theme here reflects different conceptual roles played by numeric types. In the case of one-byte integers, for example, in some contexts the full 0-255 range represents meaningful values (e.g., color), whereas elsewhere only a subset of this range is actually in effect. A ready example is enumerations (a.k.a. enums) which might use **unsigned char**s as the base or "carrier" type but only define, say, ten values. Most instances of the underlying type therefore do not match any enumerated value; the **char** (i.e., 8-bit) type is employed simply because most systems do not have types smaller than one byte (one cannot take the memory address of "half a byte," for instance). At the same time, the numeric details of how enums are encoded can sometimes become consequential even for application-level code. Consider the relatively common situation where an enum type — together with some other data — is encoded into a single (4-byte, say) integer where the enum is assigned a specific bit-range (if there are ten enumerated labels, say, the bitmask must allow four bits for the enum). A programmer would immediately know to left-shift the **enum**'s value (cast to an integer as needed) by the requisite number of bits, but could easily forget that the left-shift applied to **char** could result in *another* **char** and therefore truncate bits on the left; the **enum** instead must be cast to *int* first (or whatever type will encode the shifted bits) even if the original base type is, say **unsigned char**.

Similarly, consider using one-byte integers (say) in contexts where the underlying mathematics reflects modular arithmetic. An example is encoding directions (such as for labeling adjacency steps for inter-pixel color comparisons): a common option is to encode the eight options from top-left, top, top-right, etc., through bottom-right, via integers 0-8. On the other hand, these directions can also be derived via coordinate subtraction, spanning the eight pairs (-1, -1) through (1, 1) (excluding (0, 0)), so procedures may need to interconvert the pair-based and mod-8-based encodings. Still another possibility is to label directions such as "northwest" or "top-left" and so forth via a distinct enumeration class, where the enumeration labels would be mapped to corresponding numbers in the 0-8 range, but descriptive terms such as "left" might be more readable than numeric codes.

A variation on this example would be that of notating directions between image-locations on a larger scale via 16 direction possibilities, incorporating those between strict orthogonal, horizontal, and vertical. Actually, the XCSD code (mentioned earlier) in some contexts recognizes 24 different directions (to accommodate variations in even/odd pixel or region counts, where the center in an image-partition might correspond to a line of regions or to the gap between two regions). The XCSD format utilizes such direction-codes in several calculations (mostly related to the proximity of regions vis-à-vis an image center) where arithmetic operations are performed on the underling numeric value (e.g., mapping a direction to its nearest strict diagonal counter-clockwise). Insofar as numeric codes represent directions, then "rotations" (via modular addition or subtraction) and modular wraparounds become mathematically significant.

Modular integer-ranges can potentially be supported directly via a strong type system allowing programmers to explicitly define a (distinct) type for integers modulo 24, say. Alternatively, one might adopt something like "modular guarded" arithmetic (and binary, e.g., shift) operations, where each operation could be given an extra parameter to serve as a modular-arithmetic context (note that equally relevant for image-analysis and other geometric contexts might be variations on modular arithmetic where zero is sometimes replaced by the modulus, or results are *subtracted from* the modulus). Some of these arithmetic details might just be encoded relatively inelegantly by if-then branches and the like ("if(result >24) ..." and so forth) but we have more flexibility on the VM side to identify various forms of modular guards (perhaps via runtime flags that can be attached to ordinary integers) and of modular arithmetic in general (e.g., zero-to-modulus swaps).

The main point here is to identify coders' rationales for employing specific types and to recognize situations where algorithms need to satisfy conditions more precise than the types alone express (an integer encodes mod-24 values, an enumeration is mapped to numeric quantities for geometric computations, such as mapping named directions under rotations/reflections, and so forth). Whereas these conceptual details might be merely implicit (or consigned to comments) in high-level source code, VMs can treat such conditions more rigorously by annotating types, or flagging runtime values, or providing variants on common operations (e.g., a modular variant on arithmetic functions).

Consider, again, integer-overflow situations: if a programmer is looping *up to* an integer maximum (e.g., 255) they might select a type for the looping variable to *avoid* overflows; so if a number i, say, is to represent one color-component and be used for a loop, the algorithm could declare i as a *short* (or a *signed short* to avoid negative wraparound) since 256 (and -1) then become expressible values, preserving the loop-termination. At this point however we have a *short* encoding a color-component whose only *conceptually* meaningful values are in the **char** ranges; if the i were used in a color-related computation it would need to be cast to its conceptually accurate **unsigned char** (aside from an extra coding step, consider how this could trip up code-review or compiler-warning sensitive processes). A more fluent solution might be to keep i itself as one-byte but cast it to **short** for the loop-test, something like

```
/* u1 is unsigned char; u2 is unsigned short */
for(u1 i = 0, ok = true; ok;
    ok = ((u2) i) + 1 < 256, ++i)
```

... which admittedly is not very attractive as C++ code. This is an example of where a VM could provide extra functionality beyond the reach of compilers targeting smaller-scoped assembly languages: consider a variation on inequality operators which would combine a (temporary) type-cast and boolean test into a single instruction. Having that form of test available in a VM opset might inspire language designers to support it with distinct syntax, so that a pattern such as "cast-then-compare" — which is more conceptually precise than using artificially larger types to avoid overflow/underflow conditions — is directly supported by high-level code, potentially making it clearer and more widely adopted.

In general, VM engineers have opportunities to identify computationally similar but conceptually distinct operations. Consider a basic increment (++) operation: this may occur in a context where the range of possible loop end-values (e.g., the size of a list) is well below the maximum expressible integer of a relevant type; but it might also occur (as in color-components) where we are looping up to that maximum value, which itself serves as the loop-terminator (this in turn is the kind of situation where overflow errors due to "perpetual-truth" can occur — comparisons of **unsigned char** against 256, say, never fail). The former use of the increment occurs in a context where there is an

a priori range setting outliers on the loop (from the first to the last index in a list, say), whereas in the second case the type itself (e.g., all possible values for one color-component, in a format like RGB32) sets a range-span. One can argue that the two uses of increment (and analogously decrement, especially in the context of unsigned types when looping down to zero) are conceptually different enough to warrant two different VM instructions, especially considering that extra care is appropriate for the second case (due to wraparound bugs).

Similar attentiveness to operations' "conceptual" significance applies to functions on number-pairs as discussed above, especially in contexts where pairs represent logically integral values (such as locations within an image). When operations have a geometric interpretation (not just arithmetic) it is possible that they may be naturally presented in several different versions for different contexts, in the same way that modular arithmetic has multiple interpretations (including geometric ones related to directions and/or rotations). A VM instruction set might therefore recognize these distinctions, providing a more flexible compilation-target and perhaps cueing higher-level language engineers to similarly acknowledge the relevant distinctions (e.g., via specialized syntax).

Moreover, operation-variants and the type which they work on are conceptually interrelated, so engineers at both higher and lower levels should be attentive to type-system concerns interrelated with conceptually-distinguished low-level operations. This chapter thus far has presented examples in contexts such as modulus, integer overflow, and number-pair types; another set of examples derives from enumerations and their encoding strategies, which we address next.

2.2 Different variations on enumeration types

In the simplest sense, enumeration types represent sets of named values, which have numeric codes (for the sake of storing instances in memory) but whose numeric representation has no particular significance. This setup, however — where enums' actual encoding carries no semantic weight — is only accurate in some contexts; elsewhere, there are various possibilities for enums bound more tightly to their numeric basis. For example, nominals representing days of the week, or months of the year, have a fixed sequence, which in turn is reflected in their encoding. These are examples of types where both named labels and a limited-range, modular-arithmetic representation have conceptual merit (consider the problem of "adding" some number of days or months, or iterating around a week, or year, cycle).

Or, consider a collection of named color-values: each of these would be identified by a label, but may also be mapped to a numerically-encoded color space, so that quantitative inter-color operations could be implemented (measuring the distance between two named colors, for example, or mapping them to grayscale). Some operations on this enum type would therefore yield numeric results (such as color-distance); other operations could map one enum to another by applying quantitative formulae (such as mapping chromatic hues to gray tones, or to brighter/darker shades). So the enum type would implement intra-type and out-of-type operations consistent with a mathematical foundation, at variance with the convention that enumerated values are string-labels without quantitative interpretations.

Another case of consequential numeric encoding which can be illustrated via color-values would be using enumerators as array indices: consider enums which labels the components of color-encodings according to different color spaces, such as RGB (the enum labels would obviously be something like {**red**, **blue**, **green**}) or HSV ({**hue**, **saturation**, **value**}). If colors (for these three-part models) are represented with three-valued arrays, then the enumerated component labels should map to 0, 1, or 2 so as to form array indices.

Also, some use-cases for enums are to introduce a collection of related labels into a programming environment, without necessarily implying that there is a single semantic or conceptual interpretation that covers each of them. For instance, it is plausible that an enum employed in the context of managing calendars (not just raw dates but schedules, appointments, and so on) would have labels for days (Sunday-Saturday) but *also*, say, *weekday* and *weekend* (in contexts such as being open (all) weekdays, or happening "every weekend"). In this case, the numeric encoding for days-of-the-week would operate on two levels: most labels would be related sequentially (Sunday, Monday, etc.) but "weekday" and "weekend" would be grouping-labels instead, signifying collections (Monday to Friday and Saturday/Sunday respectively).

A plausible implementation for such a type would be via bitmasks: assign the numbers 1-64 (increasing over powers of two) to the seven weekdays (i.e., each code has exactly one bit set, in positions 1-7) and then assign the codes 31 (the bitmask of all five weekdays, assuming the lowest value is "Monday") and 96 (the bitwise combination of Saturday and Sunday) to "weekday" and "weekend," respectively. Presumably, this kind of encoding would be most natural in a contexts where bitmasks could be employed to represent various combination of days (e.g., coding "Monday and Wednesday" as the bitwise merger of the Monday and Wednesday codes). Of course, this is only feasible in contexts where enumerations can be cast to integers (whether explicitly or implicitly).[3]

Mainstream programming languages evince different patterns with respect to mapping enums to numeric values. In C++, for example, although we can specify the *byte length* of an **enum** type by requesting a particular underlying type (e.g., **enum class e : unsigned char** forces each enum value to be represented by only 8 bits), enums — at least "strong" **enum class** ones — do not automatically *cast* to integers. They can, however, be manually cast; or, integer-style arithmetic semantics can be added *to* enums via operator overloading (in the XCSD code, the `ENUM_FLAGS_OP` macros expand to implementations of operator overloads which make most arithmetic operations available to enum values, for enumerations that require them). With other languages (Java, for instance) utilizing enums' numeric values is more difficult, whereas elsewhere (e.g., C#) conversion to integers is automatic.

This variation between languages reflects inconsistency across underlying semantics: enumerated nominals have different degrees of conceptual connection to an underlying arithmetic space in different contexts. There are, indeed, at least six possible options (we're proposing non-standard terms for the final three items here, but hopefully

[3] Meanwhile, note that bitmask-encoding would introduce a further calculation step in contexts where days of the week are sometimes also given by numbers. In response, some code might prefer to use sequential numbers and assign, say, 8 and 9 to "weekday" and "weekend." Any procedures inputting a days-enum would therefore need to check for these two idiosyncratic values and branch accordingly. This may be imperfect style from a viewpoint that code which has numerous conditional branches is harder to maintain and understand (compared to mostly "branchless" programming). On the other hand, using a "bitmask" encoding would entail extra procedures being implemented (for converting the 1-64 values to their 1-7 equivalents, perhaps via a "`ffs`", or "find first bit set" operation, almost equivalent to taking a base-2 logarithm). Moreover, relying on numeric values to ascertain details through the enum — "on Wednesday" is not a matter of testing one enum value against *Wednesday*'s for *equality*, but rather for nonzero bitwise-or, because "weekday" includes Wednesday (in this sense bitmasks make certain day-related logic very concise) — forces the enum codes to have specific values, which might feel dubious from a type-theoretic perspective according to which enums should be proper "sum-over-unit" types. In effect, two different intuitions about "elegant" (and maintainable/understandable) programming style are in tension, as far as restricting branching *or* preferring enum types not bound to precise underlying values. general point is that these kinds of tensions tend to occur in contexts where enums span concepts which (albeit interrelated) have a variety of semantic interpretations, like *Wednesday* versus *weekday*.

the exposition will furnish rationales for the vocabulary chosen):

Raw nominals Collections of named labels ("nominals" in the statistics sense) which are be assigned numeric codes simply for purpose of computational representation, but where the actual numbers have no special meaning (an exception could be allowed for a label used as a "default" or fallback when a specific value is missing, which may be coded with a zero to coincide with defaults in other settings); this is the case that, technically, best matches the type-theoretic understanding of enumerations (in terms of summing unit types — those with exactly one instance — so their sum is a collection of instances which have no internal structure, at least apart from the means to distinguish one from any other).

Cyclical or sequential enumerators This would cover cases where labels correspond to entities that have an implicit order ("ordinals" in statistics) so we can speak of one label coming "before" or "after" another in the sequence as conventionally understood. In some contexts we could also recognize "differences" (as in, Friday is four days past Monday) and wraparound (after Sunday we return to Monday, for instance), which in turn implies modular arithmetic.

Enumerators as *de facto* numeric constants A different use-case applies to labels which have a specific numeric code because they are intended to give a recognizable name to an actual numeric value. Of course, in some cases important numeric values which can be assigned descriptive names are independently introduced as constants, which does not involve a specific enumerator list — for instance, C/C++ has macros such as `SHRT_MAX`, `INT_MAX`, `INT_MIN`, and the like which define maximum (respectively, minimum) values for different integer types; these are global constants which have some obvious logical interrelationship, but there does not appear to be any benefit in grouping them together as a distinct *type* whose specific purpose is to identify just these values. On the other hand, consider a situation where we want to create a list of `Latin1` codes for characters which have specific meanings in some formal context (e.g., the list of whitespace glyphs: space, tab, new line, form-feed, carriage-return). Here it is plausible that a procedure might accept a type which expresses a member of such a list (implementing actions for a code-generating grammar, say) while also each label needs to embody a specific constant value (we want the enumerator to translate directly to a corresponding ASCII and Unicode number), which in a sense combines the sorts of contexts where global constants would be preferred and those more amenable to proper enums. If indeed an enum is chosen to represent the constants as a *type*, the numeric values would of course have semantic significance (on the other hand, most quantitative operations between two labels would not make sense — there is no meaning to adding one ASCII code to another, say — so enums in this case should not necessarily be provisioned with a full suite of arithmetic operators).

Isolated bitmask enumerators Names which do not have intrinsic mathematical interpretations but which are assigned numeric codes built around powers of two, for the sake of notating combinations of such nominals via a single number. This kind of encoding is common for labels which represent "flags," with the possibility of merging multiple flag-values or "activating" multiple flags at once. For a geometric-flavored example, suppose we want to specify alignment options relative to a bounding rectangle: the actual labels recognized by the enum might be *left*, *top*, *right*, *bottom*, and *center*, but these labels could be used in combinations such as "top-left," using the bitwise-or merger of codes for *top* and *left* to indicate alignment according to both of these directions. We call enums in this case "isolated" to suggest that *within the label-list itself* each nominal has a numeric code with no intrinsic meaning; it is only for creating *numeric* representations of combinations (like **top** | **left** as a mathematical construction) that encodings' binary structure become significant.

"Fusional" bitmask enumerators We propose this terminology to express cases where some enumeration labels represent combinations of other labels. The difference between this case and the prior one is that here combinations are not only numeric *expressions* which can be interpreted as simultaneously declaring multiple nominals at once, but instead labels which are part of the enumerative collection itself can be assigned mathematical values which translate to sets of other labels. The weekday/weekend case would be one example, given a bitmask encoding such as proposed for the weekday example; or, suppose we extend the "alignment" example just cited such that **TopLeft** and other combinations are actual parts of the **enum**. Types in these scenarios have the ability to form labels for commonly-used collections of other nominals; consider, say, the type **Qt::TextInteractionFlag** (part of the Qt application-development framework) which defines settings for how a GUI control displaying natural-language texts allows for user actions (selecting, copying, following hypertext links, and so forth). In addition to several independent/isolated labels there are two bitmasks which are part of enumerator list: **TextBrowserInteraction** (which defines the most common fusion of interactions appropriate for controls playing the role of text browsers) and **TextEditorInteraction** (which, similarly, sets the defaults for text editors — which differ from browsers in that text may be modified within the control, while it is being displayed). Of course, other flag-combination may also be used; but providing labels for the *most common* combinations helps to clarify the specific conceptual status of those fusions specifically.

Free-form or arithmetized enumerators This case would cover situations where numeric values associated with enum labels are significant, but for multiple or intersecting reasons not covered by the above items. Relevant examples can include cases where mathematical functions are applied to enum-values in some algorithmic contexts. Earlier we mentioned a 24-valued "direction" code employed internally by XCSD; were these codes to be expressed via an enum, they would need a specific mapping to integers so as feed the enums directly into certain XCSD-specific algorithms (related, for example, to the process of ordering image-regions based on proximity to the center and, secondarily, by clockwise progression *around* the center). For technical reasons (the details are not especially relevant here) these directions are coded by *signed* values in the range -1 through 22; there are specific computational reasons for this range in particular, indicating that the numeric codes are intrinsically associated with corresponding enum labels (it's not a matter of assigning numbers essentially at random, or merely in an increasing sequence to honor ordering and/or cyclicality). Other "free form" examples would come from situations where codes may be assigned with an eye to bitwise fusions, but not all labels fit that binary pattern. The above "alignment" examples can have variations which fit this last case. In Qt, the **Qt::AlignmentFlag** enum gathers — into one type — the labels for vertical (**AlignTop**, **AlignBottom**, **AlignVCenter**, and **AlignBaseline**) and horizontal (**AlignLeft**, **AlignRight**, **AlignHCenter**, **AlignJustify**); and also for *one* named combination (**AlignCenter**, which fuses **AlignVCenter** and **AlignHCenter**). In addition to these labels, the list includes a special flag which is only applicable for text in natural languages with a right-to-left writing system, and "mask" labels for convenience — specifically, a mask which extracts the *horizontal* and another for the *vertical* component of an alignment code; those masks would be used mathematically to split a single code into two labels (indicating that the labels need specific numeric values to make the binary operations work properly).

As the Qt alignment example shows, sometimes enumeration types have labels serving different conceptual roles: the Qt type essentially combines horizontal and vertical options, plus a special supplemental flag, and a bitmask for convenience, into a single label-set. Proper numeric encoding is needed to orchestra the interplay of concepts and values involved here. More to the point, this example shows that expressing multiple concepts through a single label-collection increases the likelihood that numeric values will be significant.

The various alternatives outlined here reflect a duality in the "conceptual" role of enums: in effect, enum types fulfill two distinct purposes — namely, the use of mnemonic or descriptive *labels* (instead of raw numbers), on the one hand, and (on the other) isolating a specific range or group of numbers as a distinct *type*. These two roles can be combined in different ways for different enumerations, which in turn raises questions about the proper type-theoretic protocols appropriate for enums. For example, how should procedures whose signatures indicate parameters of an enum type work with inputs whose numeric values do not match any nominal label? Should compilers reject code where there is no algorithm to prove that a certain input conforms to the specific list of values associated with enum labels? Or should it allow values that can be formed from those matched to labels via boolean **or** operations (merging their respective bits) but reject others? Or, perhaps, map anomalous values to one "default" label? Moreover, we can distinguish cases where a compiler can verify that a value will *fail* via either of those constraints, as opposed to cases where the details are uncertain — e.g., the enum may be obtained via serialization, or some other external source which may (or may not) endeavor to restrict values to "meaningful" options. These issues are also reflected by how enums are used by procedures which take them: if they are fed to **switch** statements, for example, then a **default** execution path could catch any values not matching named labels.

In short, the enumeration mechanism is actually a combination of multiple semantic patterns, and these differences tend to get reflected at the point where procedures take inputs of enum types (rather than their underlying numeric types). In the VM context, procedures could therefore indicate what "flavor" or enum type they expect, through some sort of flag or annotation. Instead of a single enum protocol, compilers (or a VM runtime) could then fine-tune enum handling based on procedures' specifications. For example, a procedure could indicate that it should *only* be called with an integer value that matches an enum value, or that (as a runtime feature) anomalous values get mapped to one specific label. Moreover, enum *types* could be equipped with different varieties of integer-casts, distinguished between contexts in which conversions would be applied. In the most restrictive case, enums would never be cast to their underlying values except for a narrow set of low-level operations (such as serialization via binary packs); in the least restrictive, enums could be freely utilized as quantitative magnitudes, subject to a full suite of arithmetic, binary, and boolean operations. In between those extremes, enum types could be annotated to accept casts in certain circumstances but not others, or to support a proper subset of mathematical functions.

This discussion has set forth some of the conceptual issues associated with enumerations, and before that with number-pair types. The examples cited here serve primarily as case-studies; ultimately, we intend to ground the analysis in VM engineering in particular. The overarching theme of this chapter's discussions thus far has been that VMs might work with type systems which have programming and design-pattern nuances that aren't obvious in "formal" type theory. High-level coding patterns are not necessarily reflected, or reciprocated, at the VM tier; compilers might "compile down" source constructions to quasi-assembly languages. But this is a design choice; it is certainly possible to plan VMs that directly represent many high-level design patterns and provide supportive compilation-targets enabling compilers to support such patterns in the languages they target. As such, we will transition to a more VM-oriented (and, indeed, image-processing oriented) focus in the following section.

3 Integrating Virtual Machines with Image-Processing Operations

This chapter so far has addressed topics, such as two-coordinate data types and enumerations, which are only tangentially related to image-processing. The current section will focus on images more specifically. When considering VM "integration" with image-processing, we refer primarily to setting up Virtual Machines with inherent functionality related to invoking Computer Vision algorithms. Presumably many VMs will implement a general-purpose Foreign Function Interface would would support calls to native-implemented graphics-related procedures (along with any other high-throughput domain, where highly optimized code in languages such as C or C++ is preferable for performance reasons). In the current context we intend to focus however on VMs which are explicitly oriented toward image-processing, where this specific problem-area is a central design emphasis instead of one group of foreign-function capabilities amongst others.

This means that the VM model will incorporate features specifically relevant to images. For example — as already discussed last section — representations of image locations and intra-image lengths, in the form of distinct numeric-pair types, could be incorporated as "kernel" data types for the relevant VM. This is a straightforward example of how basic choices for VM design could be influenced by image-related use cases. We will present other examples below.

3.1 Exposing GUI Functionality

As outlined in Chapter 20, a central focus of VM functionality in these chapters involves applications "exposing" features for third-party components or plugins; within this context an important area of functionality attends to GUI state. In computer vision, a straightforward example of GUI interop would be showing intermediate stages of an image-processing pipeline. The OpenCV library, for example, provides an **imshow** procedure that displays an image in its GUI window. Typically this procedure is called in conjunction with specific OpenCV algorithms in order to visualize those algorithms' outputs, particularly while the pipeline is initially being formulated and fine-tuned.

For example, the algorithm in question might be a feature-detector which looks for "keypoint" locations (that match a specific point in an image), straight lines (matching extended, relatively crisp edges or visible lines), contours (matching closed curves, considered to enclose a 2D area) or "superpixels" (relatively small image-regions of large homogeneous interior color, also 2D). These alternatives correspond to zero-dimensional, one-dimensional, and two-dimensional features, respectively [15], [37], [14].

In most scenarios feature-detection is not intended to transform an image or to produce intrinsically visual data; instead, sets of keypoints, lines, contours, or segments/superpixels are subject to further statistical analysis. For instance, "landmark" keypoints may be used for image registration — aligning multiple images (MRI scans,

say, or photos taken with a head-mounted camera that are stitched together to form 360° "panoramic" tours) by locating pre-identified keypoints common to all images in a series. Landmark keypoints may also be used for segmentation (by connecting certain keypoints to form polygons outlining or containing a Region of Interest).

Whether via this technique or others (e.g., superpixels or closed contours) segmentations are then typically utilized to isolate a desired foreground region and analyze its color and/or shape (metrics such as its color average, or the eccentricity of an ellipse containing the region) which in turn may yield object-classification or other "semantic" interpretations of image-data — consider software which records traffic patterns by isolating visuals of cars in the context of street cameras.

Image-data generated via this kind of pipeline is typically entered into a database or in some other manner tracked analytically, so insofar as human users access these results it would be in the form of spreadsheets or other structured, largely textual formats. However, when initially formulating a Computer Vision pipeline engineers often need feedback concerning intermediate processing stages, so algorithms typically feature-detectors can be paired with operations to produce secondary images visually summarizing the algorithm's performance. Keypoints, feature-lines, contours, and superpixels may be visualized by drawing markers at key-point locations or highlighting lines and superpixel/contour boundaries with pronounced colors (distinct reds, yellows, light blues, and so forth, selected to avoid confusion with actual colors in the image). These graphics are then overlaid on the original image to represent an algorithm's results pictorially.

Such intermediate images are not analyzed themselves (they are not technically part of a pipeline) but provide performance/accuracy feedback.[4] In other contexts, intermediate graphics that *are* part of a workflow proper — such as blurred/simplified or grayscale images analyzed in lieu of an initial picture, so as to reduce noise or allow for restricted-channel analysis — might likewise be previewed during development, to help programmers conceptualize how the pipeline operates.

In short, an intrinsic demand of Computer Vision components involves rendering intermediate images visually even if those visuals are not analytically part of a workflow. Insofar as image-processing capabilities are part of a full-fledged desktop-style application, Computer Vision routines can leverage GUI features to show intermediate images, potentially with interactive capabilities allowing users to examine results in greater detail. For example, keypoints might be represented on images through boxes that receive mouse-events independently from the larger picture (recall the **QGraphicsScene** comments from Chapter 20), providing for instance context-menu options specific to one keypoint. In OpenCV, each keypoint detected via SIFT (Scale-Invariant Feature Transform) and similar computations have multiple corresponding parameters reflecting their surrounding image-context.[5] Keypoint attributes are an example of how visualizing Computer Vision workflows may involve interactive displays with more functionality than merely graphing images decorated with feature-depicting annotations (like highlights for feature-points and superpixel-boundaries).

This example also points to the potential complications involved in applications "exposing" GUI functionality for image-processing. The case of *previewing* an intermediate-stage graphic may involve nothing more than a file path for the desired image. Once the presentation involves context menus and intra-image interactions, however, the interop between a GUI host and an image-analysis (semi-autonomous) component becomes more elaborate.[6] Context menu options might furthermore be linked with procedures available in the *analytic* component, so the front-end would need to register "callbacks" deferring functionality back to the client; there would be a bidirectional conduit for metaprocedure invocations (recalling terminology from Chapter 20) instead of a client requesting one host feature (as would be the case with showing an intermediate image non-interactively).

In short, both components would need to expose respective functionality according to a mutually-implementable protocol. For reasons we have discussed, situations where components "expose capabilities" along these lines are a good example of leveraging Virtual Machines to model, fine-tune, or dynamically extend data-sharing protocols.

As a general rule, Computer Vision algorithms are largely self-contained, in that code needed to run the actual mathematical analyses — once an image for processing is loaded into memory — may be grouped together into a single library without often calling external procedures. On the other hand, image-processing workflows will typically need to defer to host applications for file-paths to load images in the first place (or some other resource-reference, such as handles into an image database) as well as (discussed in the prior paragraphs) for visualizing analytic results. Because of the sheer diversity of image-analysis methods, applications should incorporate Computer Vision in a flexible manner, with the option of adding new kinds of analysis over time (partly because different methods are better suited for different image series, depending on their coloration, occlusion effects — and in general how crisply pictures show foreground/background boundaries — textural complexity, and similar variables that can affect the accuracy of a pipeline). In this sense we can speak of "exposing" functionality in two directions; image-processing components share their analytic capabilities, while host applications expose GUI and filesystem routines, and so forth.

The prior discussion has considered interop from the host application's point of view, but we will now transition to image-processing components themselves. As a case-study we will consider the "XCSD" format which was formulated partly in conjunction with developing material for these chapters. This discussion will indicate the rationale for XCSD's approach to color-processing, memory-layout, and other implementation details, but this is not a technical documentation of XCSD intimating that this is necessarily a superior representation or paradigm. Instead, this chapter employs XCSD mostly as a case-study for implementing new image-processing functionality in a pre-existing host-application environment.

3.2 Extending Host Applications with Image-Processing Workflows

As is implicit in this discussion, the form of connections between applications and image-processing components which we will emphasize here involve hosts *calling* Computer Vision algorithms provided by external libraries, perhaps mediated through Virtual Machines. A different notion of "overlap" between VM and image-based engineering would be dedicated VMs for *implementing* Computer Vision algorithms,

[4] Many segmentation and feature-detection algorithms, for example, have multiple parameters that can be adjusted, and programmers will often experiment with different settings in the context of a sample image for which the desired analysis is known ahead of time, so summary visuals signal how well the algorithm matches that goal.

[5] These data structure allow keypoints occupying the same ground-truth location to be tracked among multiple images, e.g. for registration: we can compile key-point sets for two images and look for points with similar statistical profiles, as if matching fingerprints; scale- and rotation-invariant keypoints in particular will have analogous profiles in multiple images whose contents overlap in some area of physical space.

[6] In a context such as examining keypoints, the front-end would need access to a data set of keypoint attributes alongside the underlying image, plus instructions on how to populate context menus over keypoints and over generic image points, respectively.

analogous to how Graphics Processing Units provide computing environments tailored to concerns such as ray-tracing and other graphics-rendering calculations. Designs for compiling image-processing code *to* special-purpose VMs, which are optimized for running such algorithms very efficiently, constitutes a separate topic, which is outside the focus of this chapter (except indirectly since calling conventions within such algorithms might overlap with calling conventions for initiating image-processing workflows or workflow steps, the latter topic fitting more within the bounds of invoking Computer Vision capabilities externally from a VM context).

Image-analysis functionality can be grouped into several areas, such as contour-detection, region morphology, texture-description, diffusion analysis, and methods related to color statistics (e.g., color histograms). For purposes of exposition, this section will discuss algorithms related to the XCSD format mentioned earlier. The basic focus of XCSD concerns how pixel-data is stored in computer memory; in particular, pixels are lined up into groups at different "tiers," or nested levels of detail. The XCSD code accompanying this chapter supports three tiers, based on powers of three; so 3×3 pixel-boxes are grouped together, as are 9×9 and 27×27. This structure is noticeably different from conventional "matrix"-based image formats, where one entire row of pixels (spanning the whole image-width) would typically be coded in memory as one pixel-run (sometimes called a "scanline"), followed by each successive row of pixels in turn. This representation is of little intrinsic value, however (apart from simply recording pixel-data; that is, the *o*rganization of this value in memory adds no further benefit) unless one is working with sub-images spanning the whole distance from left to right margins. In XCSD, pixels are stored in memory such that 3×3 boxes represent one contiguous chunk, and can be accessed via raw pointers (rather than copying from scanline-based data to temporary buffers); then, on higher tiers, the same applies to 9×9 and 27×27 groupings. Figure 1 shows several graphics which can be produced from XCSD images summarizing some aspects of the XCSD format and memory-layout.

At the uppermost scale on this "tiered" system, the 27×27 boxes are called "tierboxes," which are aligned in memory from the center of the image outward. The XCSD code uses calculations based on the Manhattan-Chebychev hybrid distance representation (mentioned earlier) to arrange tierboxes such that contiguous pixel-runs correspond to meaningful image regions, specifically, diamonds, octagons, squares, and rectangles centered on the overall image-center. This provides a rationale for tierbox memory-layout, clarified below in more detail. *Within* each tierbox, pixel-runs are stored on 9×9 and 3×3 scales. The goal is to increase the likelihood that a particular cluster of pixels of analytic interest can be accessed as a raw pixel-run (via pointers to start and end memory) without needed multiple copy operations. Also, XCSD allots memory space for data associated specifically with boxes on each level, from 3×3 to 27×27. This allows algorithms to consider the image on coarser-scales, avoiding features on the individual-pixel level that could be statistical "noise," as desired. Pre-computed values such as color-averages can be recorded in the image data along side pixel-level information itself.

The multi-level representation just described gives rise to the notion of "subdivision" indexing, wherein individual pixels are accessed via their location in nested three-by-three groupings rather than absolute coordinates. Such indexing is useful in contexts where we may want to examine image-features present at different level of detail; shifting focus from the pixel-tier to the 3×3, 9×9, or tierbox levels simply involves dropping a subdivision-indexing coordinate. Also, XCSD allots space for encoding image-data of varying provenance (e.g., texture codes or region-of-interest masks) alongside channels, both at the pixel level and higher tiers (referred to as "extension" channels). The XCSD format can therefore record analytic data internally, such as masks on different tiers placing units on that tier within (or on the border of) one or more regions-of-interest.

The last two paragraphs have explained some features and rationales for XCSD, and mentioned the origin of concepts like "extensible" channel-systems and subdivision indexing, reflected in the format's name. The purpose of this chapter is not to analyze XCSD itself in detail, but rather to adopt it for case-studies in VM integration with image-processing. We will do this through the lens of algorithms associated with XCSD, which hopefully serve as representative examples of how VMs can interface with algorithms related to image-formats in general.

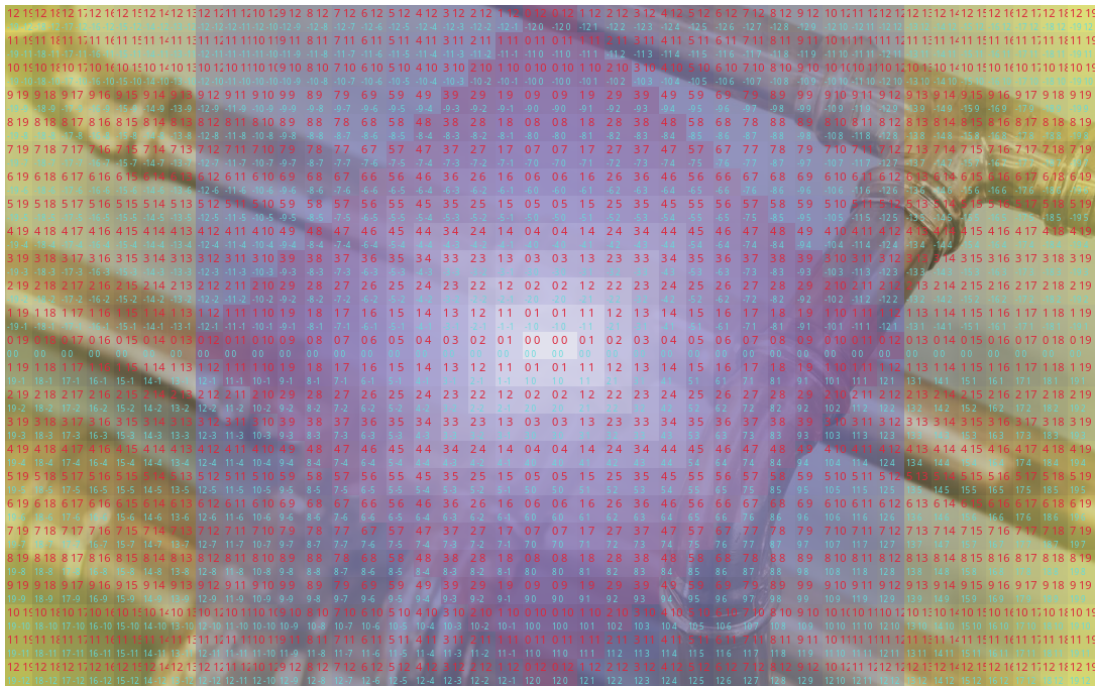
3.3 Manhattan/Chebychev distances and "Black-Grey" grids

Earlier we mentioned that XCSD uses a "hybrid" combining Manhattan and Chebychev distance metrics. For the sake of discussion, we'll call this combination an `MCH` distance-representation (not a metric, *per se*, because the result is a two-valued pair rather than a scalar). Formally, the `MCH` pair between $\{x, y\}$ and $\{a, b\}$ is $\{M - C, 2C - M\}$, where M and C are the Manhattan and Chebychev distances (see the formulas from last section).[7] Visually, the `MCH` merges how the two component metrics "count steps": whereas the Manhattan counts only orthogonal steps, and the Chebychev freely counts *either* orthogonal and diagonal steps, the `MCH` represents a path *first* along a diagonal and then, "turning" by 135 degrees, orthogonal steps outward. The raw-number pairs can be supplemented with one of eight directional indications as needed (between two points the diagonal-then-orthogonal path can have four initial directions and two possible further directions for the orthogonal component). Picture `MCH` pairs as the sum of a diagonal vector (sloping at 45 degrees from the relevant line) end-to-end with an orthogonal vector.

In order to document the relevant mathematical properties of `MCH` we will introduce a couple of auxiliary constructions related to discrete geometry. The main results we are leading uo to are summarized in Table 1 with respect to finding locations equidistant from some point. The math involved here is barely above grade-school level, but there is some combinatorial trickiness in fully enumerating all possibilities for distance comparisons, so this presentation hopefully does not seem more technical than the subject warrants. In particular, it is significant to contrast locations (especially in image-processing) which represent *pixels* themselves and which represent boundary-points *between* pixels; this comment would then generalize to overall image-regions. If a picture has odd pixel-sizes both horizontally and vertically, the exact center is one pixel; otherwise, it is the gap *between* two pixels (for an even/odd mixture) or four (in the both-even case). This motivates the following modification to the notion of discrete geometry on an integer grid or lattice:

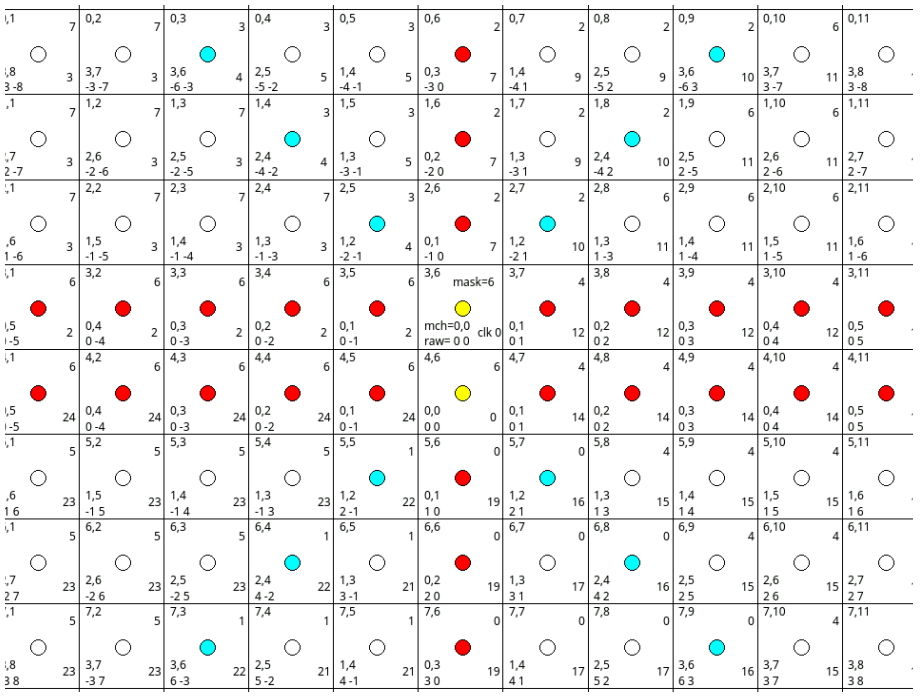
Definition. Call a "black-gray" grid a littace with lines grouped into two classes, *black* and *gray*, where each horizontal (respectively, vertical) black line is surrounded by two gray lines, and vice versa (i.e., the two colors are interspersed). The "points" on such a grid would lie at intersections between lines, and given the color-differences there are four kinds of

[7] Actually, XCSD uses a slightly different pair we'll call `MCH′` equaling $\{C, M - C\}$, but this text's version of `MCH` is more intuitive for exposition. They are readily interconvertible: `MCH′` $= \{innersum(\texttt{MCH}),\ first(\texttt{MCH})\}$ and `MCH` $= \{second(\texttt{MCH′}), innerdifference(\texttt{MCH′})\}$.



(a) Teirbox indices, from center

(b) Classifying tierboxes by foreground boundary inclusion



(c) Metrics obtained from black-gray lattice constructions

(d) Gridlines

(e) Visualizing internal tierbox data

Figure 1: Visualizing components and quantities for the XCSD format

XCSD includes various tools for visualizing how images are stored internally according to the XCSD protocol, including [a] seeing tierbox indices (data is stored from the center outward); [b] foreground and region-of-interest boundaries can be defined at the tierbox-level; [c] reviewing internal metrics XCSD employs to locate pixel memory addresses (according to subdivision-indexing); [d] viewing subdivision-lines into (by default 27×27) boxes; [e] generating detailed high-zoom summaries of images showing data for each pixel and each of the subdivision-scales.

points: double-black (intersections of two black lines), gray-black, black-gray, and double-gray (reading the horizontal color first, then vertical). Unless stated otherwise, we will refer to `MCH` distances as comparisons between two intersection-points on a black-gray grid. In this context "points" discussed

without further qualification mean "grid points," i.e., intersections between grid lines.

The purpose of this kind of grid is to address certain mereogeometric or mereotopological situations where distances can combine entities of differing dimensions. Consider a checkerboard pattern; imagine placing chess pieces either *inside* board squares (corresponding to double-black grid points), or along edges *between* squares (gray-black mixed) or at corners where four squares intersect (double-grays). The black-gray construction distorts actual geometry (if we picture gray and black as evenly spaced lines then colors do not affect the dimensions of points or lines involved, so the mereogeometric interpretation is not visually implicit, but the point of black-gray grids is for discrete geometric properties, such as distance, which do not correspond to Euclidian space anyhow). We are particularly focused on the following:

Definition. Call an MCH *cycle* on a black-gray grid to be a collection of double-black points equidistant, by some specific value, from a central point, within the context of a rectangle (possibly a square) centered on that point. If the stipulated center-point is not double-black, calculate the shared MCH distance according to the double-black point adjacent to the center (either two points alongside, for black-gray or vice-versa, or four points around, for double-gray) which is nearest to each candidate double-black point.

Note that two MCH pairs are "equal" if both components are equal (they may differ in the vector-directions). If two MCH pairs are *not* equal, there is no way for them to represent the same *Euclidean* distance (assuming we treat the diagonal and orthogonal vectors as immersed on a Euclidean plane) so MCH provides a reasonable alternative to Euclidean distances with respect to ordering point-pairs in terms of their respect distances, as mentioned last section, or grouping those representing the same Euclidean (and therefore MCH) distance.

Observation. The MCH and Euclidean distances are similar with respect to ordering point-pairs into lesser, equal, or greater internal distances. In particular, (i) for two MCH distances which internally sum to the same number of steps, the MCH and Euclidean orderings coincide. Also, (ii) strict equality between distances matches between both representations, and (iii) if one MCH coordinate is held constant, or if both either simultaneously increase or decrease, the changing coordinates will increase or decrease with Euclidean distance.

Proof Suppose we have a normal (one-color) grid of evenly-spaced orthogonal lines separated by a fixed unit, so we can assign them integer coordinates. One formula for expressing Euclidean distances in terms of MCH would be (1) $d = \sqrt{x^2 + 2xy + 2y^2}$, where x and y are the MCH *orthogonal* and *diagonal* components. Assume the pair $\{x, y\}$ is modified by the scalars j, k, respectively, for $|j| < x$ and $|k| < y$, with d^2 correspondingly modified by h. The equation for $d^2 + h$ would be $(x + j)^2 + 2(y + k)^2 + 2(x + j)(y + k)$; substracting the square of d as in (1) from this yields (2) $h = j^2 + 2k^2 + 2xj + 2yj + 4yk + 2xk + 2jk$. It is obvious that h is nonnegative when j and k both are. If both are nonpositive, note that $|2xj|$ would be $\geq j^2$, and likewise for $|4yk|$ and $2k^2$, given $|j| < x$, $|k| < y$, so the nonnegative terms (the squares) in (2) are more than matched by terms with negative or zero values (all the others), concluding that h in this case has to be negative or zero. These two cases (where j and k are not contravariant) confirm (iii) — whenever x and y are both made larger, or respectively both smaller, their corresponding Euclidean length will be larger or smaller, as expected. The remaining question concerns mixing positive j with negative k, or vice-versa. Suppose $\{x, y\}$ and $\{x + j, y + k\}$ represent the same total number of steps, i.e., they sum to the same amount, so $j + k = 0$. Then, plugging $k = -j$ into (2) yields (3) $h = j^2 - 2yj$. In this case $|j| = |k| < y$, so the second term is greater in magnitude than the first, which forces h to be negative when j is positive; when j is positive, h of course will be as well because both terms in (3) then end up positive. This proves that k and h are covariant, which conforms to MCH ordering: a greater diagonal term, for the same total steps, is defined as a larger MCH value; since h has the same sign as k, the Euclidean distance will increase or decrease alongwith the MCH, demonstrating (i). For $j + k$ not zero, we can rewrite (2) to include $(j + k)$ terms, as in (4) $(j + k)^2 + 2(x + y)(j + k) + 2k(y + j)$. Setting $s = j + k$, (4) becomes $h = s^2 + 2s(x + y + k) + 2k(y - k)$. Let $a = x + y + k$ and apply the quadratic formula to solve for $-s$: (5) $a \pm \sqrt{a^2 - 2k(y - k) + h}$. If $h = 0$, we can move the bare a in (5) across the equals sign, and let $b = -a$ (note $b^2 = a^2$) to derive (6) $(b - s)^2 = b^2 - 2k(y - k)$. Expanding (6)'s left side and rearranging yields (7) $s^2 = 2bs - 2k(y - k)$. This is only possible if s and k are both zero, because by stipulation s, b, k, and y are integers, and a $\sqrt{2}$ would fall out of the right-hand-side upon taking the root when converting s^2 to (integer) s alone. Since $s = j + k$, $s = k = 0$ forces $j = 0$, meaning that $h = 0$ implies $j = k = 0$ — and, also, the converse is obvious from (2) — proving that two MCH values can yield the same Euclidean distance only if they are identical, which confirms (i).

In short, MCH is a reasonable replacement for Euclidean distance in many contexts because there are few situations where point-pairs have greater MCH but less Euclidean distance, or vice-versa (as shown by the above proof, the only scenario where the two representations differ in this sense is that an MCH pair which sums to fewer steps, but has greater diagonal component — i.e., $k > 0$ from (5) in the proof — may yield a greater Euclidean length than an alternative MCH pair with more steps but less diagonal). Calculations such as those related to MCH cycles are not affected by those discrepancies.

Observation. An MCH cycle can have 1, 2, 4, or 8 points.

Proof Every point in an MCH cycle will have some pair $\{diagonal, orthogonal\}$ shared by all points, which will differ by direction. Since there are eight possible directions attributable to each pair, a cycle can have eight different points. We then have to identify cases where cycles will have *fewer* points. Note that if either or both coordinates in the MCH pair are zero, then the distinction between diagonal and/or orthogonal directions goes away, eliminating some points. The 1-point case corresponds to zeros for both diagonal *and* orthogonal components on a double-black center (so there is only one point, the center itself). A double-zero on a black-gray or gray-black center represents a length-two cycle (because we consider the two double-black points around the center) while a double-zero MCH for a double-gray center engenders a length-four cycle. An MCH with one zero and one nonzero component can have four directions (since both a nonzero diagonal and an orthogonal with *no* diagonal supports four) and therefore represents four distinct points in a cycle (this is another length-four case). Moreover, for any MCH with nonzero *orthogonal*, it is possible that half of the points in a full cycle (within a sufficiently large rectangle) are excluded because they lie outside the bounds of the actual rectangle associated with the specific cycle.

Note that there are several factors influencing the length of an MCH cycle: the dimensions of the bounding rectangle (and the degree to which one side is longer than another, causing some cycles to "lose" points); whether the center is double-black, double-gray, or a combination; and the presence or absence of zeros in the desired MCH distance. An orthogonal zero, for example, results in four points along two diagonals without an additional orthogonal projections which could extend either horizontally or vertically; as a result, the cycle can have only four (not eight) points, but also it cannot be contracted due to width/height discrepancies in the bounding rectangle.

The reason why XCSD employs MCH cycles is to itemize tierboxes which are equidistant from the image center. As mentioned earlier, XCSD computes a memory-layout according to which tierboxes closest to the center occupy lower memory addresses, with pixel data encoded in contiguous raw memory expanding outward from that center. This "expansion" is understood to progress through MCH cycles. The MCH format has a intrinsic ordering based first on the total length (adding orthogonal and then diagonal) and then, for paths with the same sum result, treating paths with fewer *diagonal* steps as shorter (diagonal steps are longer from a Euclidean perspective). Given any collection of points around a center, we can partition the set into XCSD cycles which have a natural ordering between one another; moreover, *inside* each

center	zeros	lengths
double-black	none	4 or 8 (depending on bounding rectangle)
	diagonal	2 or 4 (depending on bounding rectangle)
	orthogonal	4
	both	1
gray-black or black-gray	none	4 or 8 (depending on bounding rectangle)
	diagonal	2, 4, or 6 (depending on bounding rectangle; complete is 6)
	orthogonal	4
	both	2
double-gray	none	4 or 8 (depending on bounding rectangle)
	diagonal	4 or 8 (depending on bounding rectangle)
	orthogonal	4
	both	4

Table 1: Possible MCH cycles and cycle-lengths

cycle the points are distributed clockwise or counter-clockwise (since the basis for separating points is alternative directions superimpose on a single directionless MCH pair). The end result is a coherent algorithm for ordering points subject to constraints that points nearer to the center should be placed before those further away.

For XCSD, the "points" are actually tierboxes (or gaps between them), but this is consistent with black-gray grids embodying mereogeometric relations through discrete/integer mathematics, without the grid being a faithful *representation* of the modeled space's actual (Euclidean) geometry. In short, the algorithms we have described in the black-gray context work also for the practical task in XCSD of deriving the proper memory-layout for image tierboxes (and, by extension, individual pixel runs).

One rationale for memory-layout oriented to an image-center is to increase the likelihood that the pixel-data for an image-region of interest can be obtained simply via pointers to the start and end of a memory-block (without needing multiple copies into a temporary buffer). Due to the nature of MCH cycles and ordering, a contiguous pixel run (assuming it is aligned to tierbox boundaries) would take on a diamond, octagon, or square shape around the image-center (or potentially a rectangle, if the length of one side of the demarcated region matches the shorter image dimension). While many significant image-regions of course will not be centered on the full image's center, there are still situations where focusing on pixels closer to the overall center is desired. For example, signals extracted via image-analysis are more likely to be significant for labeling, classifying, or matching images when they emerge from an image's central region more than its periphery. The color spectrum or histogram weighted toward the center typically bears more significance than further from the center, for example, and similarly for prominent contour-shapes, textures, diffusion processes, and so forth. And colors near the center are more likely to be characteristic foreground-tones than ones closer to the edges.

Also, more simplistically, thumbnail or preview image-summaries might need to strip away peripheral content, even if the image is also scaled down. This points to potential benefits of center-oriented memory layout. Copying pixels (even multiple pixel-runs, e.g., one per scanline) is not a very time-consuming operation, especially in the context of full-scale image-analysis, where executing Computer Vision algorithms could easily take much more time than setting up an initial pixel-matrix. However, some image-processing may occur in contexts such as image-databases where performance delays may noticeably affect usability, in context where it could be necessary to work through hundreds or thousands of images in response to one user query/action. In these kinds of contexts, avoiding even the relatively minor step of pixel-copying can improve performance, in situations where one wants to construct thumbnails for hundreds of pictures (matching a query, say), or perform basic analysis on a large image-series.

For these reasons XCSD is designed as a format particularly suited to hosting images in a database, where basic operations such as image-compression, calculating dominant colors, and building image-thumbnails can be highly optimized. It's also true that the MCH representation, which we have described here in the context of memory-layout, can have other benefits. For some analytic algorithms, for example, having a near-Euclidean distance formula on an integer grid may be useful. Furthermore, memory-organization based on MCH cycles includes (partly to calculate the layout ordering in the first place) numerous data-points applicable to each tierbox, such as distance and orientation against the image center, information which may have some meaning in certain Computer Vision contexts. For example, the weight given to some color, texture, or diffusion scale within the bounds of a given tierbox — or perhaps as compared between tierboxes — might be affected by the tierboxes' distance from center, or from image central-orthogonal or diagonal axes, or angular distance from some analytically significant line (consider a trendline through control points suggesting a foreground region). Some contexts may prefer angular distances measured via integers (e.g., via MCH) rather than Euclidean floating-point approximations.

These points are suggestive of potential MCH use-cases; our goal here is not to advocate for MCH *per se*, but simply to justify the claim that this is the *sort* of construction that we might want to engineer in an image-processing context. We therefore claim it is a reasonable example for discussions within image-analysis contexts. That is, we propose to use MCH-cycle ordering and related algorithms as case-studies for VM engineering which encapsulates Computer Vision capabilities (or an interface to them).

3.4 XCSD operators as representative image-processing functions

The above discussion has hopefully motivated some details concerning algorithms specific to XCSD. In particular, the ordering of tierboxes around an image-center is determined by MCH cycles for progressively larger MCH values. These and related calculations give rise to a series of functions giving basic quantitative information about tierboxes, such as:

Translating tierboxes from row/column coordinates to MCH pairs This results in a measurement of the basic distance of the tierbox from one central tierbox — or one of two or four central tierboxes, depending on even or odd tierbox row and column counts (the even/odd details correspond to gray/black, black/gray, or double-gray cases reviewed above for MCH cycles on black-gray grids).

Mapping index numbers for a desired center-originating ordering of tierboxes to MCH values Such an ordering can be seeded

by expanding outward according to Chebychev distance, forming equivalence classes of Chebychev-equidistant tierboxes. Within each such class, MCH distances expand outward from rows and columns passing through the image center (or centers; again, tracking even-odd effects as with black-gray grids) toward diagonals. This algorithm has to be adjusted for the case where the Chebychev distance is greater than half the shorter image dimension (in terms of tierbox-counts) so that the MCH cycle is restricted to the image's longer axis.

Calculating tierbox positioning within each MCH cycle The choice of how to order tierboxes *within* one cycle seems mostly arbitrary. In XCSD, we chose to order counter-clockwise from the left (for images with landscape orientation) or top (for those with portrait orientation, i.e., height greater than width). reasoning was that if an image *were* to be chopped from only one side, it may be slightly more likely to cut from the right or from the bottom.[8] Shrinking an image by deleting tierboxes toward the end of contiguous memory requires no copying data, of course, which provides a rationale for placing more-likely-expandable tierboxes later in memory than earlier.

Identify tierbox position within its MCH cycle in terms of image-directions and octants In general an image can be divided into eight wedge-shape slices (i.e., octants) and members of a cycle for MCH without zeros will lie in one such octant (those *with* zeros may lie on the lines between octants, e.g., diagonals). This is one example of an algorithm utilizing the "direction" enumeration we mentioned in the last section. As explained there, XCSD's notion of direction has 24 options, corresponding to length-8 and length-4 cycles (the latter representing pure diagonals) and then distinct codes for pure orthogonals (MCHs with diagonal component zero) distinguishing even-odd permutations at the center point (as modeled by different gray/black grid combinations). Code needs to translate formulae expressing how different MCH pairs translate to one of these four direction-codes (this furnishes an example of where enumerations need precise numeric values, mentioned last section, because these algorithms make some use of modulo and bitshift operations when "collapsing" the 24 directions to accommodate different MCH cycle-lengths).

Map "subdivision indexing" to orthogonal coordinates Inside tierboxes, XCSD recognizes indexes across 3×3 and 9×9 tiers, mirroring the actual memory layout. For example, the center of each tierbox is located via code "555." Obviously, sometimes it is convenient to access points or box-regions in a more conventional x/y or row-column format, so one requisite calculation is converting subdivision location codes to orthogonal distances against tierbox corners and those of the image overall.

Compute color averages within box-regions of different tiers One rationale for the subdivision system is to establish data structures for 3×3 and 9×9 boxes (as well as tierboxes) which proxy the collection of pixels within them. If an image is compressed by a factor of three, of course, pixels "collapse inward" so 3×3 regions *become* pixels on the smaller scale. Given a desired color-averaging procedure, XCSD precomputes color means so that "compressed" images are represented within the larger image's memory, in case algorithms intend to work with the smaller-scale images directly, or to analyze on several scales at once. Apart from color-averages, data from different sources (e.g., region masks) can be attached as an "extension channel" on multiple scales. For example, a numeric code designating one region of interest (in the sense that a given pixel or box-region lies on that region's interior) can be notated as a channel for individual pixels, or boxes on the 3×3, 9×9, or 27×27 tiers, or any combination thereof. The same applies to codes representing textural patterns, annotations, or any other superimposed data.

Applying algorithms to sets of tierboxes (or other region-boxes) There are numerous scenarios where subdividing images into smaller regions can produce more efficient algorithms. This includes pixel-based computations (e.g., global color histograms) where pixels' contexts vis-à-vis surrounding pixels is not considered, so images can be subdivided simply for purposes of parallelization. Other analyses — such as those involving *local* color histograms — explicitly employ image-subdivisions as a unit of computation. Finally, some forms of image-processing are based on a lattice of "seed" points, which in a "subdivision indexing" scenario can readily be provided as the centers of tierboxes, or 3×3s/9×9s.

All of these are operations that would be relevant for images encoded via XCSD, potentially used with a diversity of front-end applications. Software components recognizing XCSD would therefore need a protocol to invoke the relevant procedures, exposed by XCSD code (yielding cross-component integration requirements along the lines analyzed in Chapter 20). The situation is analogous with many other image-formats: most image-processing applications will accept images in multiple formats, and many formats have specific operations or information unique to that format, rather than generic to any image-data. This raises the question of how software may be extended to support new formats that may come along.

Supporting formats entails accessing data proper for Computer Vision algorithms, but also incorporating format-specific GUI capabilities. For example, XCSD includes code to model individual tierboxes via distinct graphics-scene objects, so each tierbox receives user events independently, with its own mouseover and context-menu effects. The earlier discussion about GUI integration mentioned basic-level interactions such as viewing pipeline-stages via **imshow**, but some image-formats will support a much more extensive suite of GUI tooling (e.g., formats which have a color-map option for channel reduction — mapping all pixels onto one of a small group of colors — should allow users to examine the specific tones in each color map). Further cases of GUI functionality will be outlined at the end of next section.

4 An example image-processing pipeline

Having defined several operations that could be exposed by an XCSD-based image-processing component, the following discussion will outline how various functions could be pieced together into a larger workflow. This summary will point to steps in the pipeline wherein data-sharing between the XCSD component and a host application might be warranted, thereby serving as concrete examples of how each side could expose feature for interoperation.

One premise of XCSD is that conventional formulae for converting images to grayscale are often flawed. The rationale for analyzing "grayed" versions of images — rather than the full-color originals — is that most statistical analyses rely on color-discontinuities. Analyses look for points in an image where there is an above-average gap between two adjacent pixels. Discontinuity, however, is an intrinsically "scalar" or one-valued concept. For example, given a sequence of numbers, a "spike" in value would correspond to a point where two sequential numbers have an unusually large difference, compared to typical adjacent number-pairs. Generalized to two dimensions, the

[8] Given that our vision tends to be oriented left-to-right and top-to-bottom; in other words, we are perhaps inclined to perceive a foreground focus leading from the center top/leftward more than the opposite, increasingly the possibility that it will be peripheral content toward the bottom/right that is deemed expendable, if the image is chopped.

same notion of discontinuity works across different directions (up, right, down-left, etc.), giving rise to a matrix of difference-measures, because any pixel is "adjacent" to several others (eight others, if we include both orthogonal and diagonal directions). However, quantifying discontinuity by value-difference only makes sense, intrinsically, when there are just two values to compare. Full-color spectrums distort such measures. In the typical RGB format, colors can be compared along three axes (red, green, and blue). If adjacent pixels have fairly large red differentials but small differences in green and blue, say, should the measure of their mutual difference be weighted more toward the larger gap (the red) or the smaller gaps (blue/green)?

To head off these sorts of questions, Computer Vision algorithms usually perform a "channel reduction," wherein the three channels of a full-color image are reduced to one single channel (which can be visualized as an image "in gray," although in principle one-channel graphics can be rendered with any one single color, such as red or blue, fading to white and darkening to black at points of greater or lesser magnitude). A simple way to reduce channels is, indeed, merely to take a grayscale version of the original image, as if it were an old black-and-white Polaroid. The single channel thereby measures color-intensity — how dark or light it is — ignoring color-hue entirely. This tactic might discard potentially useful color information, because very different colors can have similar grayscale values. For instance, medium-red and medium-blue would, if juxtaposed, strongly suggest some ground-truth discontinuity, but they could both map to almost indistinguishable grayscale values if they are similarly mid-range in intensity (neither dark nor light). For these reasons, some image-processing frameworks prefer alternative channel-reduction techniques.[9]

The XCSD format is designed to support several alternative reductions to one-channel. One tactic employed by XCSD code involves "toroidal" color space, which is related to HSV. A toroid is a solid torus; topologically, a torus is the result of one circle rotating around a central point, tracing its own circular path in the process. A point on a torus surface can therefore be identified by two angles, one measuring how far the "rotating" circle has gone around its anchor point, and one fixing a spot on the rotating circle itself (of course, we initially have to choose an angle to represent "zero" both on the surface and at the center — the latter axis would point out in space away from the center of the torus "hole," like a rod threaded through some bagels on which they hang by their centers). In HSV, the *hue* dimension is typically understood as circle-like (embodying the "color wheel" whose opposing colors are green/magenta and blue/yellow). For any given hue, the colors generated by varying "value" and "saturation" in the HSV model naturally form a triangle, with the pure color fading to white along one side, and decaying to black on another, while the white-to-black side spans progressively darker shaded of gray. Near the midpoint of the triangle would be a "muddied" version of the pure color, appearing as the blend of that color with grayish tones.

More broadly — setting aside computational image-processing for a moment and considering human vision — people appear to experience color "thematically," rather than purely chromatically, in that we are receptive to the optical effects through which color-shades become visible as well as to actual color tones. The experience of color is not primarily oriented to monochrome color-spreads, but rather (we might say) colors as "themes," wherein one thematic color carries expectation of some textural and lighting-effect variation. A straightforward example of such a theme might be a canonical foreground color; or, we may have an image-foreground composed of multiple objects with their own predominant color, each thereby becoming distinct themes for the overall image (but still encompassing textural patters across objects' surfaces). In nature, it is rare to encounter either solid-hue patches or quality pigmentation; more common are surfaces that reflect light with some scatter and inconsistency, altering the primary color-value and modulating apparent shades as the eye scans laterally. At the same time, this is an intrinsic feature of visible appearance — it is actually oversimplistic to describe surface appearance *only* as a matter of color in terms of chromatic channels (red/green/blue, etc.). Vision is about obtaining information optically, and all facets of light waves' interaction with objects are candidate data-sources.

If a surface's primary color is red, say, we would not expect that object to appear uniformly and smoothly red, like a half-Polish/half-Indonesian flag. Instead, "redness" would be phenomenologically experienced as a distribution of relatively reddish tones, shading to black/white or variants (browns, grays) suggesting shadow, illumination, and the muddying effects of rough (material) texture. Collectively these are sometimes called "textural effects" and analyzed through Computer Vision through texture analysis [39], [3], [12], [46], [22], [50], [30], [33], [11], [43], [44], [25], [16], [40], [27], [53]. Because the material coarseness (or, inversely, shininess) is an intrinsic physical property of objects — akin to red/green/blue color — effects such as texture and glare/shadow may even be modeled via distinct channels, alongside chromatic tones themselves (other potential supplemental channels may be glossiness, also called "specularity," and transparency). Image-analysis cannot examine physical materials directly, of course, so texture-related qualities need to be inferred computationally, and color-models might be evaluated in terms of how efficiently they feed algorithms which address these optical dimension of color-appearance.

In practice — returning to variations on the HSV model — most real-life colors do not appear with pure hues; instead, because of glare/shadow effects and/or material texture (which causes light to scatter off images, so that the visible color at given points has a grayer appearance than the underlying pigmentation would suggest) a patch of color (representing one evenly-colored surface) would typically span several HSV triples (with varying amounts of white, black, or gray mixing with the primary color). From this observation we can envision an HSV triangle being centered at a midpoint between pure gray and the pure primary hue, a point from which all other parts of the triangle can be defined in polar coordinates (i.e., as a rotation away from a line between the pure hue and pure gray). This rotation is *additional* to the hue's own rotation in the color-wheel, so the overall color space is defined by two angular coordinates, which is why the space could be described as a toroid.[10]

Given this construction, we can define a color-comparison metric in terms of two rotation angles and one radius length (the "rho" measuring

[9] We admit that this outline is a little oversimplistic. Some grayscaling algorithms are more sophisticated than simply averaging out red/green/blue measures; there are ways to incorporate information about how our eyes see color, so that some hues "feel" brighter or darker [2], [31], [21], [26], [32]. Equal saturation amongst the three red/green/blue channels does not yield equal psychological experience of dark or light, and these differences can be incorporated into "realistic" techniques for color-to-gray conversions. Also, even a relatively small gap in gray-measure will be detected by most Computer Vision algorithms, e.g., to identify boundaries between different image-segments. So in some sense even grayscale formats incorporate *some* hue-data, indirectly. Nonetheless, differently-hued segments which have similar psychological brightness-valence will still be experienced as distinct color-patches. Likewise, small grayscale difference may complicate algorithms because it may be ambiguous whether such differences result from textural or light/shadow variation within one relatively homogeneous color-patch or actual segment-partitioning. In other words, even a more-refined grayscaling algorithm is subject to processing errors for similar reasons to simpler gray-conversion formats.

[10] Unlike a toroid proper, such a space has a "pure" white-to-black line which yields identical colors for every hue, which distorts color-distances in ways that do not match a toroid shape; however, actual color-distance formulae within the toroid should be weighted anyhow, so the internal geometry does not match a toroid in Euclidean space to begin with.

distance from a neutral grayish manifestation of the pure color, with larger rho values being pure-color, pure-black, or pure-white). By weighting these parameters the HSV triangle could be distorted (into something like a disk, more in keeping with the toroid shape), with the goal being to derive a color-distance metric which accommodates how empirical texture and glare/shadow effects can alter *apparent* color-distance (see also [49]). Whereas a visual illustration of the global HSV space would appear like a cylinder, the resulting variation on HSV results in something like a toroid (hence a "toroidal" color model).

An assumption behind this model is that because even "solid" colors (vis-à-vis an object's surface pigmentation) will appear sometimes lighter, darker, or grayer, these textural/lighting effects should be compensated for when computing color-distance. Exactly how to weight toroidal parameters is a matter for future research; the existing XCSD implementation employs a straightforward balancing mechanism which tries to compress distances for near-black and near-white shades and expand distances for purer (high-value, high-saturation) tones.

Code for XCSD applies a toroidal model to derive several channel-reduction strategies. One computation involves estimating a good reference color typical of an image's *background*, and another canonical color (also called a "pole" in XCSD's code) for *foreground*. Via distance from these colors, XCSD computes a probability that any given color belongs to a foreground-object or background-area. This probability is a scalar quantity, so it serves as a reduction to one-channel, one which is typically noticeably different from conventional grayscale.[11]

For the case-study workflow discussed here, XCSD estimates foreground/background poles by assuming that foreground colors are likely to be featured near the center of an image, and background colors closer to the margins (particularly on top). Such heuristic may not comport with all images, but it is a good place to start. The workflow then leverages XCSD's "subdivision" layout to grab dominant colors from histograms local to tierboxes both near the image-center and top-periphery, averaging out these two cases to obtain working foreground/background poles.[12]

Having assigned foreground/background probability to all pixels, this workflow then reifies the probability scalar as a channel and can invoke different feature detectors. The illustrations here are based on a BRISK (Binary Robust Invariant Scalable Keypoints) detector [23], which yields a fairly extensive set of keypoints almost all of which are positioned within regions of the image which, on visual inspection, clearly belong to the foreground.

4.1 From Keypoints to Superpixels

Once useful keypoints have been identified, most pipelines will then generalize keypoints to image areas, either enclosing or bounded by relevant keypoints. In the current workflow, a superpixel segmentation (see, e.g., [47], [19], [17], [1], [36], [24]) of the image is performed and superpixels which encompass some of the BRISK keypoints are identified; because the keypoints are localized near the foreground, the sum of keypoint-containing superpixels then serves as a reasonable approximation of the foreground as a whole. This is not the only way to use zero-dimensional keypoints; a more sophisticated analysis might consider keypoint attributes so as to more rigorously screen for those in the foreground proper.[13] For the sake of discussion, however, consider the more immediate calculation where the overall set of keypoints is treated as a point "cloud" roughly distributed through the foreground. Intuitively, keypoints at Region-of-Interest boundaries would seem to be the most valuable, because they would directly yield useful segmentations; however, complex/occluded images may require more circuitous techniques, because feature-detectors will generate *too many* keypoints. To compensate, one option is to actively seek out algorithms that identify large numbers of features and then apply heuristics to separate those belonging to the background and foreground (this also applies to two-dimensional features, e.g., edge-detectors: the graphics in Figure 6c intimate how detected edges can be more concentrated in the foreground than the background, which provides options for leveraging kernels that detect seemingly superfluous edges, rather than trying to retrict algorithms to contour-boundaries).

Returning to the example, visual inspection suggests that the BRISK results do indeed match the foreground well, but the workflow will attempt to calculate this result systematically. To do so, the code loads a data structure (defined in an XCSD-specific language) which describes a series of image-transforms that rotate and slightly skew the image, then blur the image to solid-color 27×27 blocks or boxes, and cancel (white-out) all boxes that are not sufficiently close to the foreground color. These operations manipulate the image so that the solid blocks may act as classifying bins, where the same transforms are then applied to the key-point set, such that each keypoint gets mapped to a specific box, which could be labeled as either foreground or background. The goal is to test, first, that the keypoints are indeed mostly found in the foreground (i.e., most keypoints map to a foreground bin) and also that the keypoints are distributed relatively thoroughly across the foreground (i.e., most foreground bins include at least one keypoint).

In combination, the two statistics just outlined confirm that the aggregate of keypoints — particularly when extended to the containing superpixels — approximates the foreground, both by *including* most of the foreground and by *excluding* most of the background. Such assessments can be explicity quantified by measuring the percentage of keypoints which get mapped to a foreground (not background) box, and also the percentage of foreground boxes with one (or more, above some desired threshold) keypoint — the higher both percentages, the more effective the workflow.

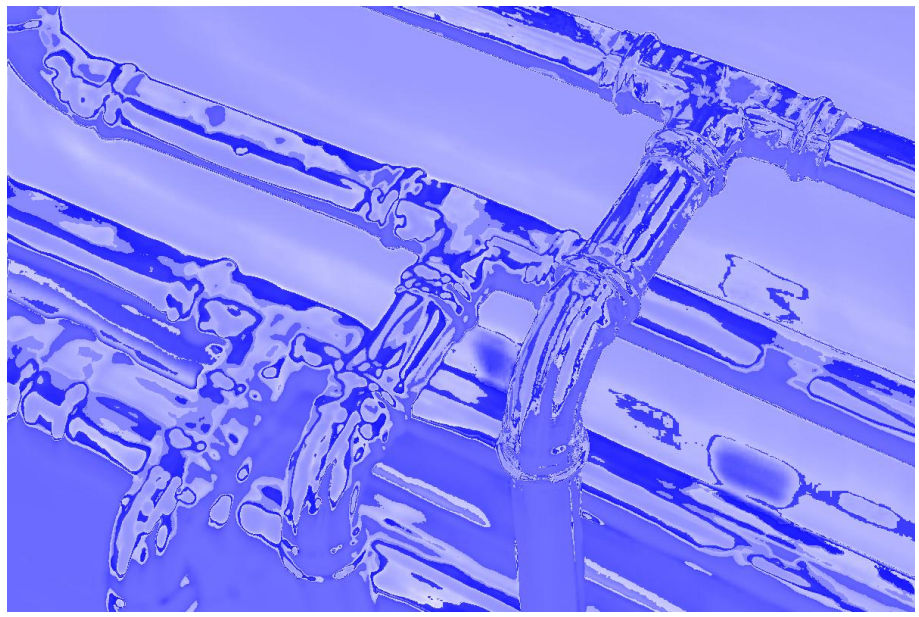
As a final detail, a different XCSD distance metric is employed to calculate "Hough" lines — one form of line-detector [8], [34], [29], [42], [45], [7] — so as to derive a good angle with which to rotate the initial image. Here XCSD dispenses with one-channel reduction and instead considers "local" color distance, considering color variation within a relatively small (e.g., 7×7) area. Unlike converting a full image to grayscale (or any other one-channel reduction), in which a useful color-distance metric should apply across the image, when measuring color-distance locally we can focus on how far apart two nearby pixel's coloration appears in the local context, without considering the

[11] The probability metric is actually a combination of two other values, difference *from* foreground and from background, with the assumption that pixels *closer* to the foreground color and *further* from the background pole are the most likely actually to be in the foreground — but unless the two poles are actually at a maximum inter-hue distance, which is unlikely for real-world pictures, the two distance-from-pole parameters can varying independently, which is why they must be averaged together (arithmetically inverting one of the pair) to yield a one-channel reduction.

[12] A variation of this pipeline would allow human users to fine-tune this selection manually, by selecting specific tier-boxes and examining their histograms. It is helpful to support interactive image-processing along these lines, even if only to fine-tune workflows eventually designed to run without human input (e.g., users might engage interactive methods to finalize parameters for an image series in the context of one or several sample images, then fix those parameters for non-guided analyses extended to the series as a whole). Here, though, the discussion will stay focused on fully-automated workflows; interactive options will be touched on at the end of the section.

[13] I.e., attempt to find statistical signatures strongly suggesting that a given keypoint is indeed part of the region's foreground (or near the foreground/background boundary, in which case such boundary-keypoints could serve as a scaffolding with which to isolate the foreground).

(a) Original



(b) Background Probability

(c) Foreground Probability

(d) Combined

Figure 2: Background and Foreground Probability Scalars for Channel-Reduction
(foreground probability is shown red-to-black, background is shown blue-to-white, and the two scales are merged in the combined graphic)

presence of other pixels elsewhere in the image.

Multi-channel color distance is complicated by the fact that distance-magnitudes may take on different meanings in different contexts; we might measure two colors as relatively far apart and then find a third color which is equally distance from the original two, leading to questions of how the respective distances should be ordered. However, consider the border between two different image-segments: there will in general be relatively little color-variation *inside* each segment, but a notable differences between sample colors in one segment and in the other. Near the border, then, color-variation on either side should be reduced, but cross-border comparisons will be much larger. So long as we only consider local neighborhoods where the latter cross-border distance clearly outpaces intra-segment color distances, we can set aside the question of how local distances compare to "global" color distance vis-à-vis the image as a whole. Applying this metric, consequently, we can isolate pixels that appear to lie near segment-boundaries.

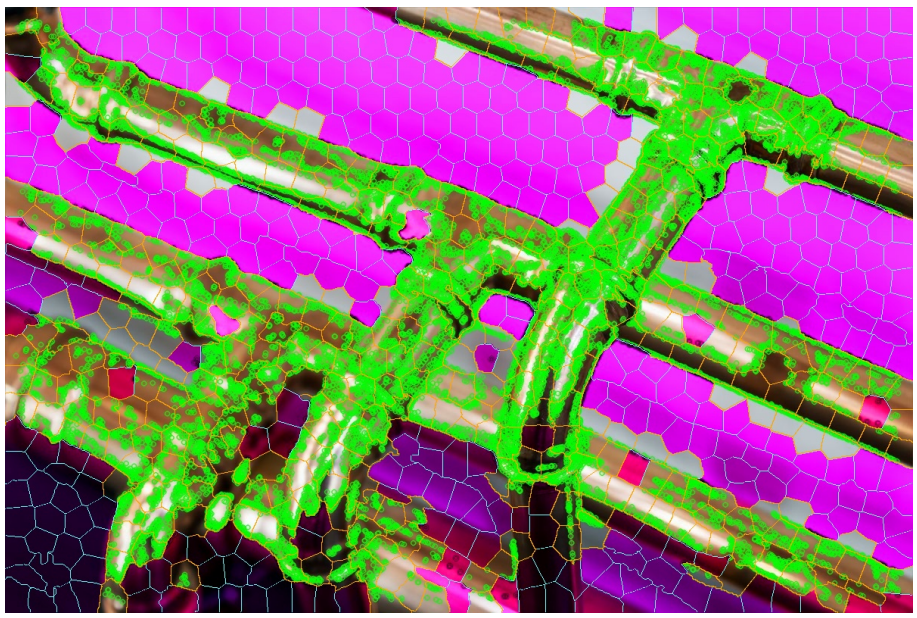
In the context of line-detection, this latter strategy for producing a single image-channel — measuring local-neighborhood inter-pixel color difference — can potentially enable inter-segment lines to stand out clearly. For the case-study described here, all Hough-detected lines have almost identical angles of inclination (they appear parallel when overlaid on a base image), which is a good indicator that the average of the Hough-lines gives an angular orientation for the image as a whole. If so, noteworthy boundaries or other one-dimensional features tend to be aligned along a specific angle (or at right angles to that) so that rotating the image by that angle allows those prominent features to be oriented parallel to the image sides — which, in turn, allows the image to be partitioned into orthogonal boxes for purposes such as keypoint-classification. In this example, such classification serves to assess the quality of the prior pipeline steps (BRISK keypoints plus superpixels), rather than being a further analytic step proper, but such summary evaluation can play a legitimate functional role within a pipeline overall (e.g., helping to produce a measure of confidence in the pipeline results).

The workflow just described is fully automated, but it still belongs in the context of an overall interactive software environment. For instance, the analyses calculate numerous intermediate parameters (canonical foreground color, global angular orientation) whose correctness affects the accuracy of the whole process, so they should be monitored. Any workflow can likewise be compared against other workflows, either using similar methods but different specific conventions (e.g., different channel-reduction formulae) or very different methods. Initiating and then contrasting *multiple* pipelines is an interactive phenomenon which must be supported at the application level.

4.2 Interactive workflows and assessments

(a) BRISK

(b) SURF



(c) BRISK overlaid on Superpixel

(d) Colormap Channel-Reduction

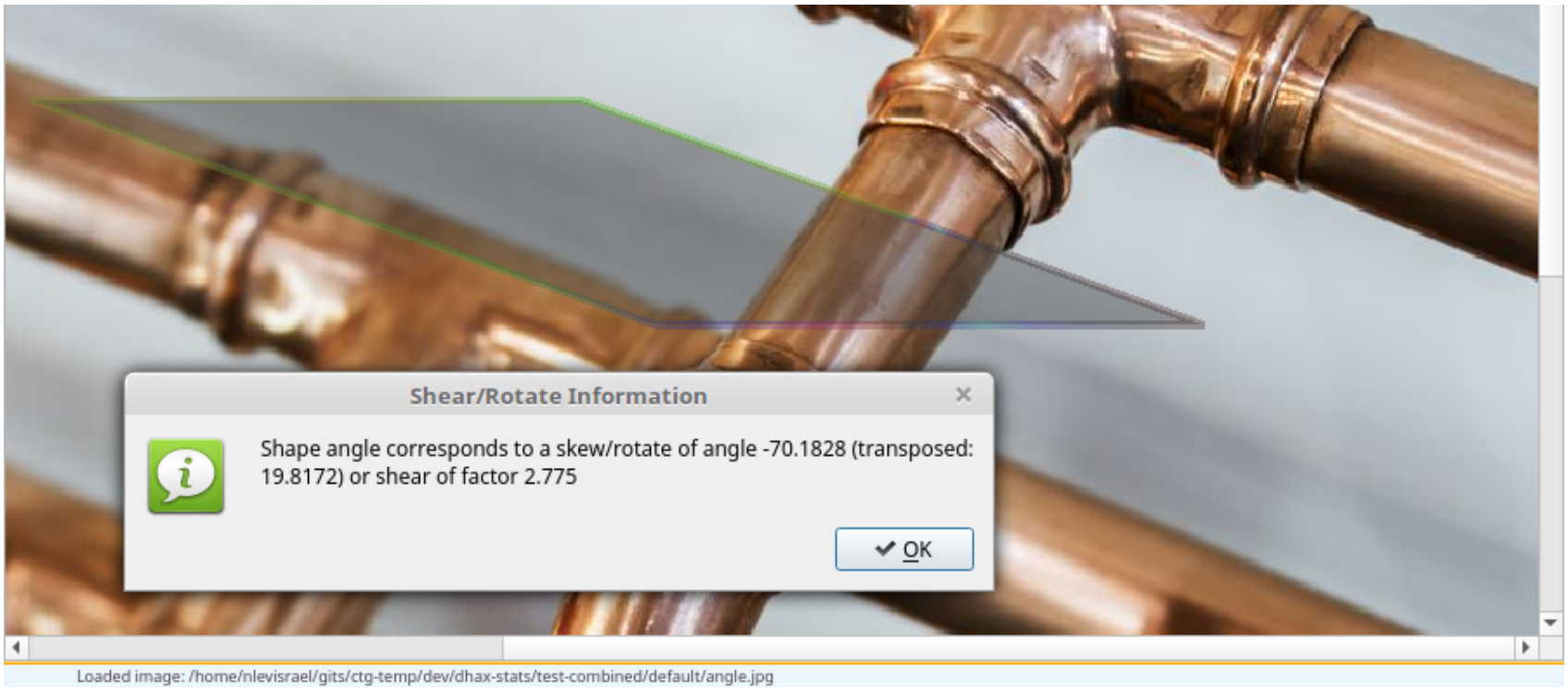
Figure 3: Localizing Keypoints to the Image-Foreground

Two keypoint detectors yielding good results for this workflow, BRISK (Binary Robust Invariant Scalable Keypoints) and SURF (Speeded Up Robust Features) [a, b]. Combined with a superpixel segmentation, the foreground could be detected by extending the feature point-cloud to the set of superpixels containing one keypoint (or some count of keypoints, beyond a chosen threshold) [c]. This superpixel segmentation uses a different channel-reduction via mapping the full set of image-pixels onto a small collection of the most frequent colors, according to toroidal distance [d].

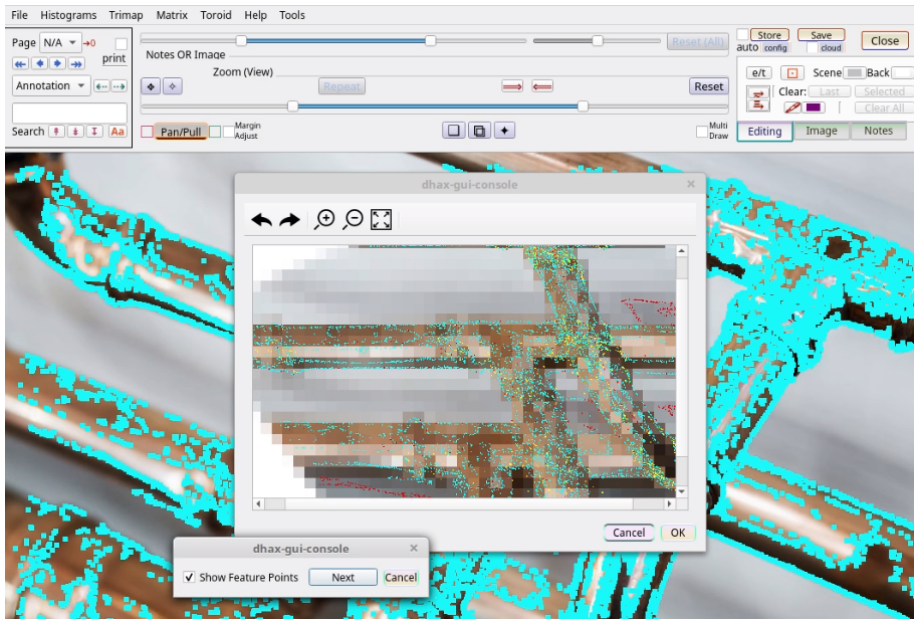
Most image-processing algorithms are designed to be fully automated, proceeding without user input. However, techniques such "interactive segmentation" (plus comparing *different* techniques, as just cited) are an exception, relying on human guidance to fine-tune a workflow's behavior. There are several scenarios where an interactive approach is useful, including training data for subsequent automated processes which involve machine learning, establishing parameters for workflows are mostly automated, achieving greater performance for image-series that are elude most Computer Vision methods, and developing protocols for testing the accuracy of automated methods. Whatever the specific rationale, interactive workflows require more complex application-level support than automated ones, because users need to act on image-graphics and workflow data through GUIs. This user-interface dimension implies that interactive image-analysis presents a representative use-case for general themes in software engineering (including VMs).

To illustrate, consider a fairly simple interactive workflow which could be juxtaposed with the automated workflow outlined above. Our goal is to isolate the copper-colored foreground from a gray background and measure the relative size of the foreground. Unlike the automated process, we can proceed without special mathematics (matrices or Gaussian kernels) or attempting to isolate the foreground with some level of detail — we only want to get an approximate cover for the foreground (what is sometimes called an "object proposal" [35], [52], [4], [48], [9], [20], [6], [18], [13], [41], [10], [28], [51]). If there is sufficient and consistent background/foreground color-difference we can take color alone as a good indicator of foreground-probability. The automated workflow tried to estimate this probability without human intervention, but in an interactive session users could manually identify representative foreground and background color, perhaps via a GUI color-picker (which reads the color under the cursor when clicking via a mouse, after the user signals an intent to temporarily apply the mouse-press in this manner). This step represents one parameter influencing the workflow's behavior (or two, if users select both foreground and background shades) and also one interactive modality that must be programmed into a GUI.

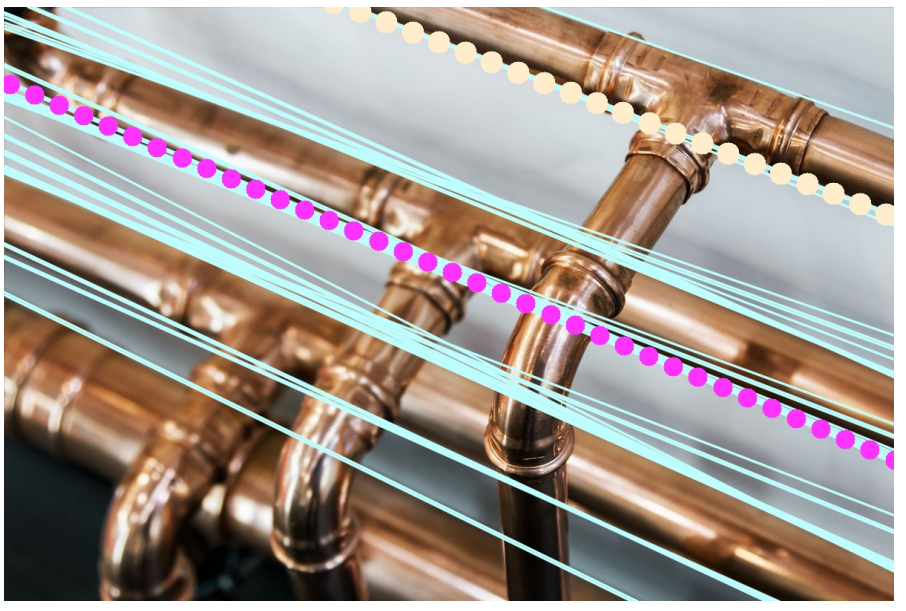
Given a selected foreground color, all pixels within some specific color-difference (this is another detail that may be set interactively — how close must that difference be and via what formula to calculate it) are mapped to the foreground color and all others to the background, resulting in a two-color image. This may not actually yield an isolated foreground, however, because the foreground-classified pixels could



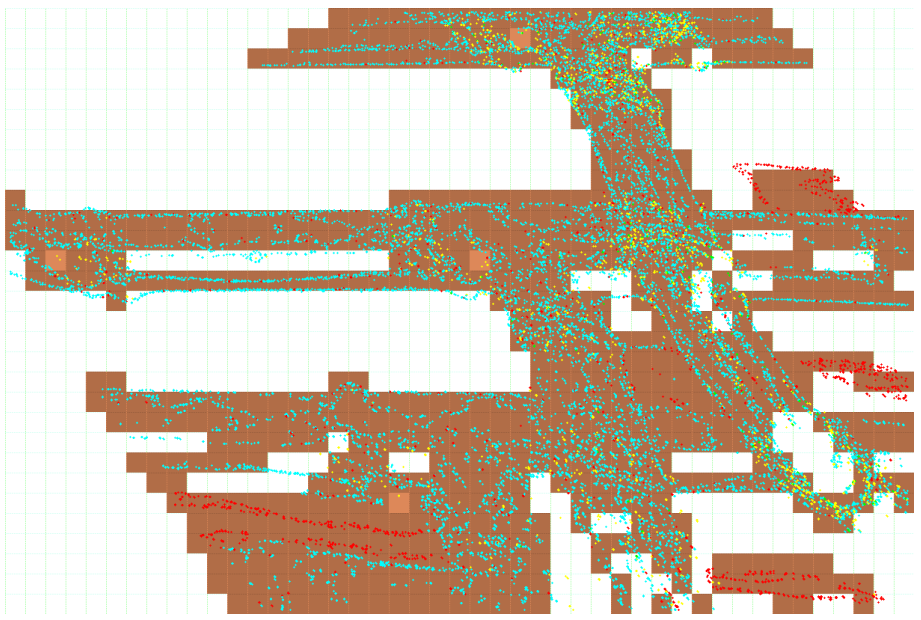
(a) Selecting a rotation angle manually



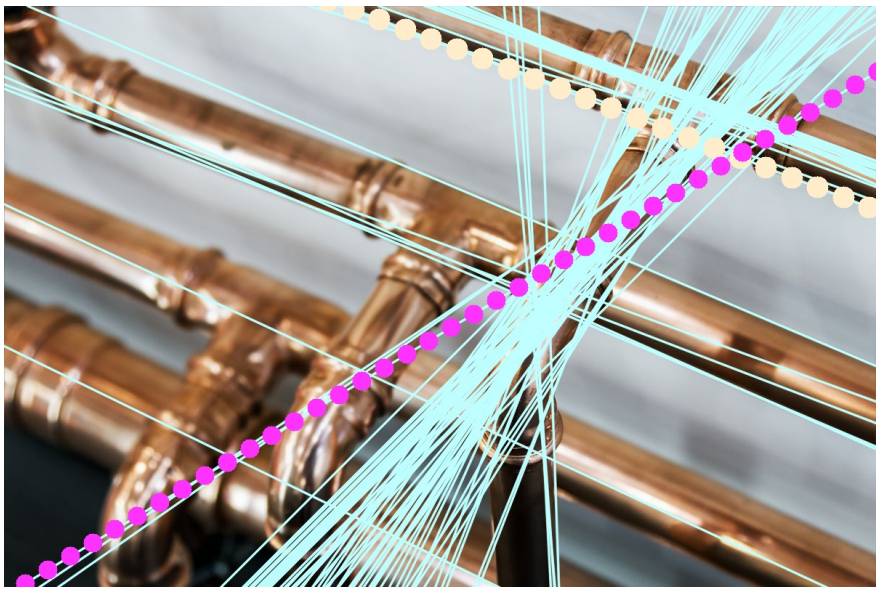
(b) GUI support for stepping through the evaluation transforms



(c) Deriving a rotation angle via HOUGH line-detection



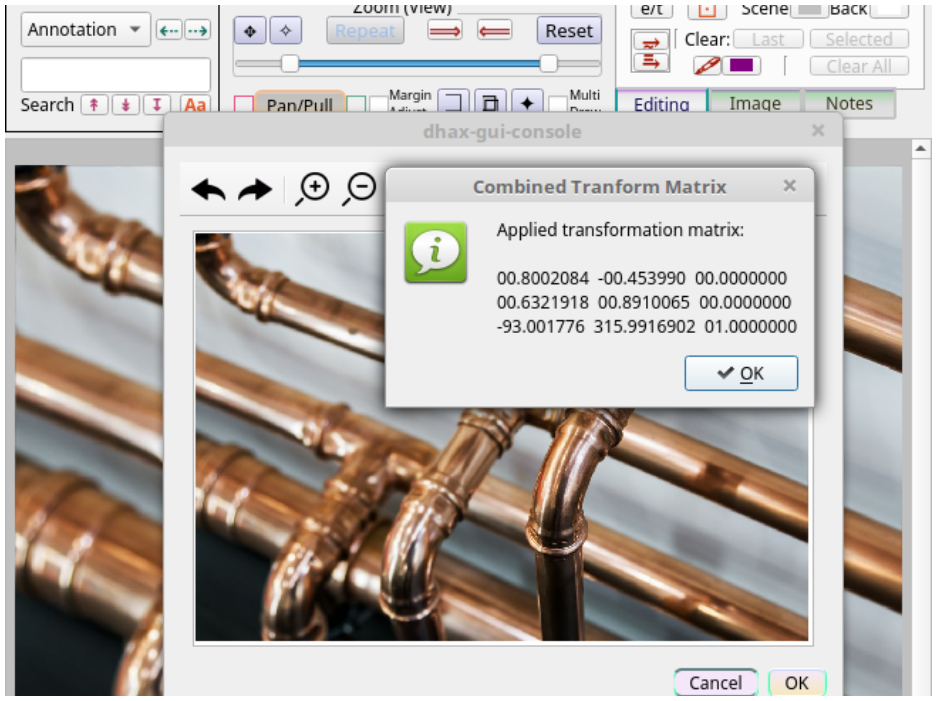
(d) Evaluating how the keypoints map to binned foreground/background areas



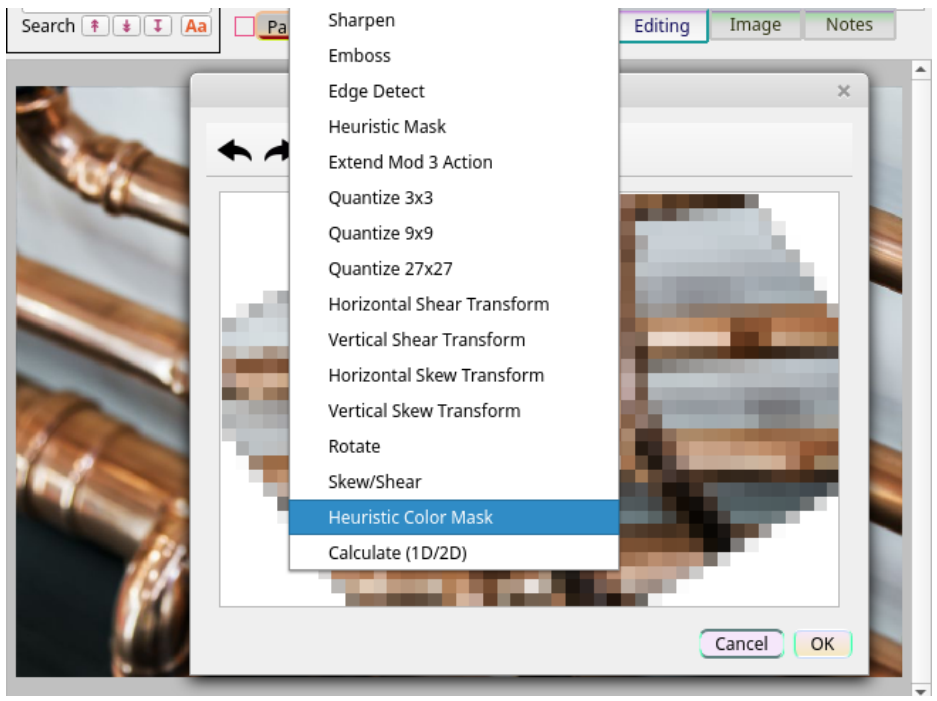
(e) The same HOUGH line detector run against a conventional grayscale reduction (average angles are magenta, target angle tan-orange)

Figure 4: Deriving a Rotation Angle for Evaluating the Keypoint Detector

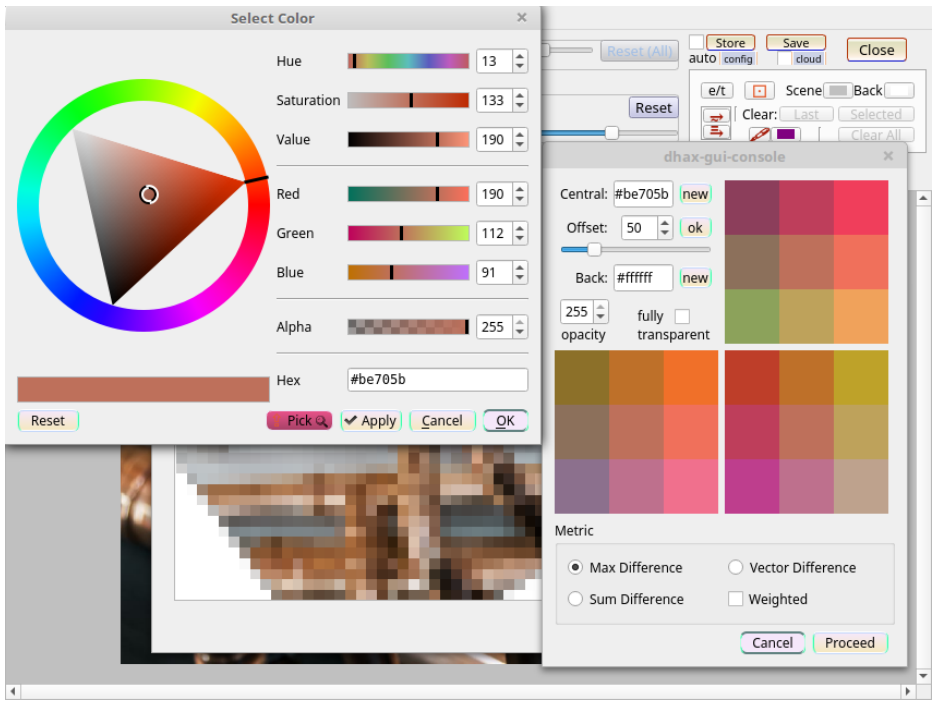
To calculate how well the keypoints match the foreground, we transform the original image to yield orthogonal boxes, where it is easy to calculate how many keypoints (if any) lie on each box. Visually, note that an automated HOUGH line-detector operating on an XCSD toroidal color space yields results comparable to manually selecting a global angular orientation for the image [a], [c]. Mapping the image to a skew/rotated and discretized box-set to quantify how closely the keypoints map to the foreground [b], [d]. Watershed methods (based on color-average inside each box, and their distance to the highest-probability foreground color) classify each box as approximating a foreground or background region. For comparison, [e] shows a duplicate of [c]'s line-detector only using a conventional (realistic) grayscale conversion in lieu of XCSD channel-reduction; the latter analysis shows an entirely different (and mutually inconsistent) set of HOUGH lines.



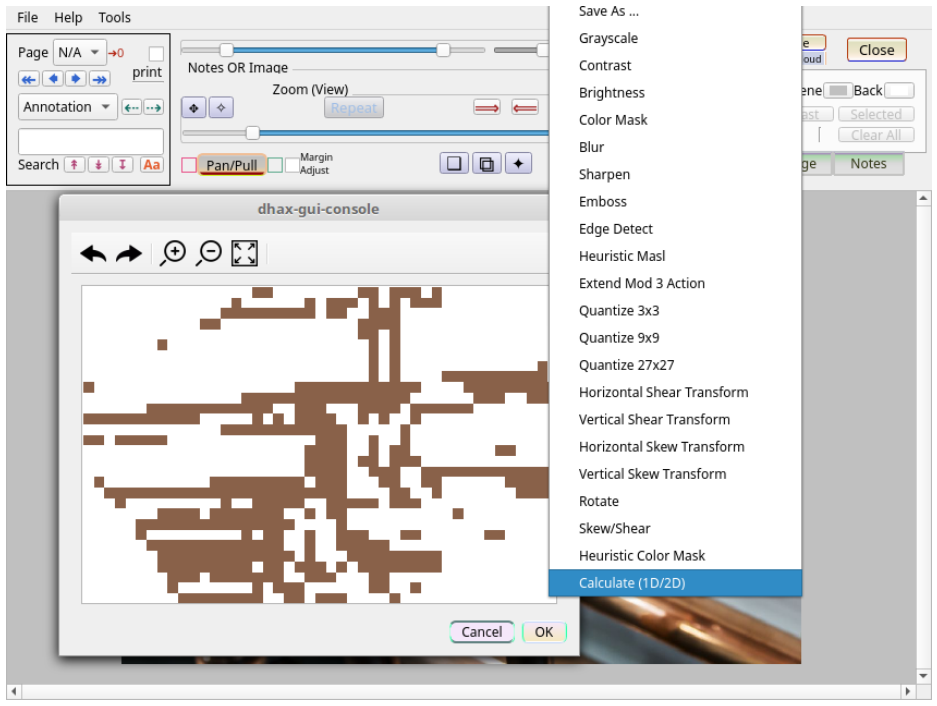
(a) Preview the Rotation Angle



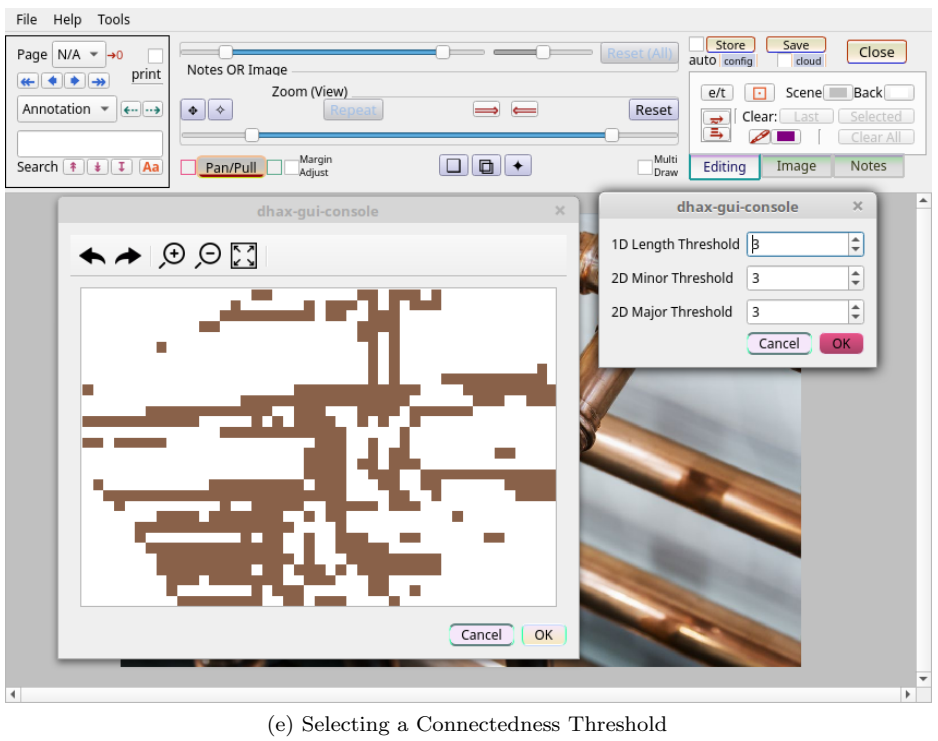
(b) Initiate a Color-Mask



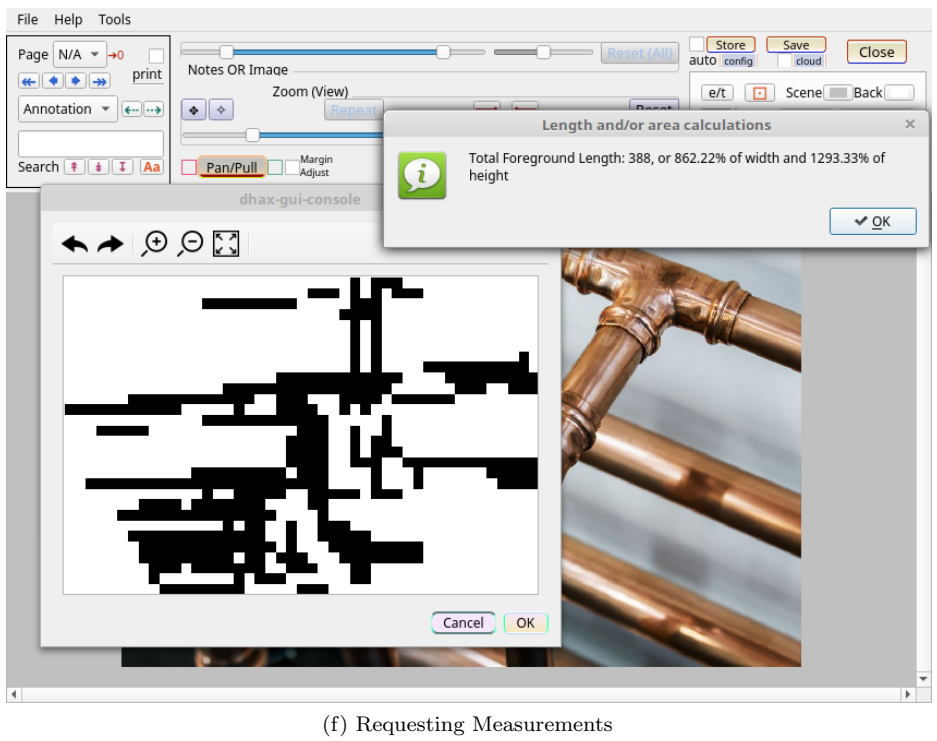
(c) Select a Color for the Mask



(d) Request Foreground Calculations



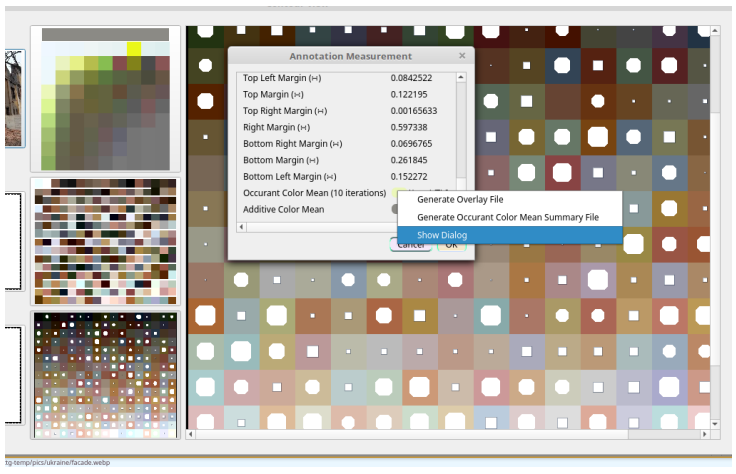
(e) Selecting a Connectedness Threshold



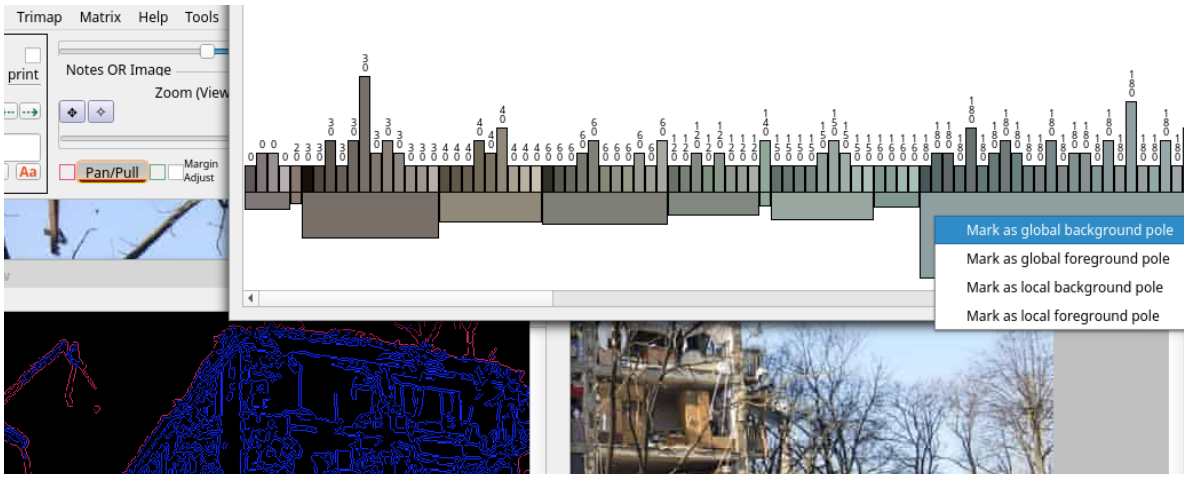
(f) Requesting Measurements

Figure 5: An Interactive Workflow with XCSD

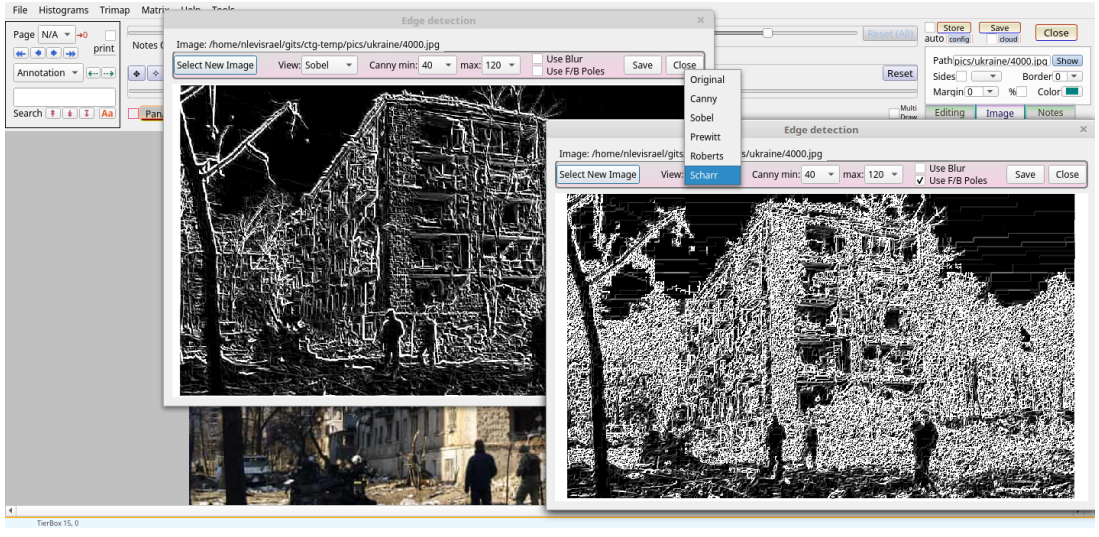
Defining an image-transform workflow interactively: first applying a selected skew/rotate angle [a]; selecting a color, difference-threshold, and difference-formula for foreground-isolation [b, c]; choosing a threshold to group connected foreground boxes [d], [e]; measuring the size of the foreground approximation against the total image dimensions [f]. As shown in the previous figure, a similar workflow can be defined to test the automated results of a keypoint-plus-superpixel workflow that tries to isolate keypoints within the image-foreground.



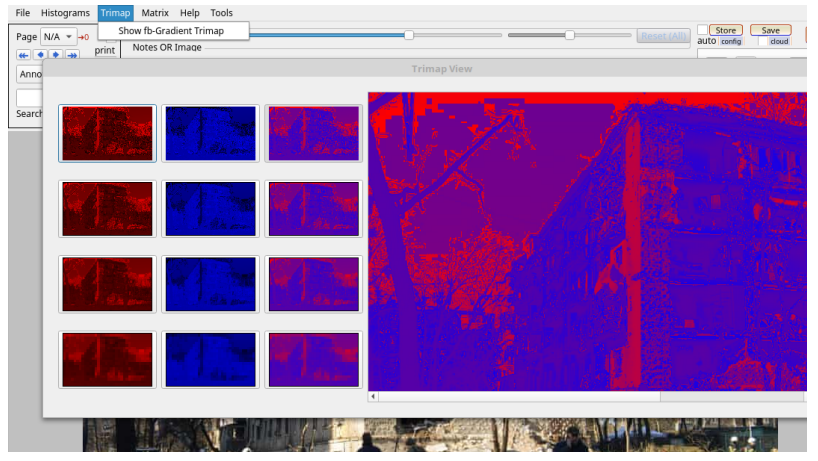
(a) Color Distribution (interactive tools to study images' most prominent colors)



(b) Interactive Histograms (local histograms to aid interactive thematic-color selections)



(c) Edge Detection (interactive GUI support for comparing different edge-detection algorithms)



(d) Foreground/Background Probability (visualizing probability scalars and their combinations at different discretization levels)

Figure 6: GUI Tools for Interactive Image Processing

Screenshots of GUI components for interactive design or evaluation of Computer Vision workflows.
Viewing color distributions and histograms [a], [b]. Comparing edge-detection algorithms [c].
Visualizing foreground/background probabilities [d].

be scattered without a well-defined shape. To counter this, we can segment the image into boxes, treating each entire box as foreground or background depending on whether the average color within that box fits within the specific foreground-difference.

In order for boxes to align with the foreground shape — rather than straddle both foreground and background, cut through by the boundary between them — we might start the workflow by rotating the image, so that most lines of the foreground, to the degree possible, run parallel to the sides of the image. This is another varying parameter — how much to rotate any specific image — and also another feature for the interactive GUI. As shown in Figure 4a, the XCSD code implements such a requirement through a rhombus-shaped annotation that users can register as a rotation-instruction (there is also an option to pair the rotation with skew or shear actions, further aligning foreground-boundaries to horizontal/vertical axes). The GUI walks users through steps of enacting a rotation, converting the granular pixel-data to coarse 27×27 boxes, testing each box for color-distance from the selected foreground, and then mapping each box to either the foreground or background color.

The last steps in this demo workflow are to isolate the foreground proper from imprecise pieces, perhaps image anomalies or perhaps indeterminate boxes which actually straddle the foreground/background line. Again, the goal is not to neatly delineate the foreground but simply to gather a collection of medium-scale boxes that approximate its size and shape. We therefore want to include only boxes that piece together coherently — forming lines and areas, rather than individual foreground-boxes dispersed within the background. This implies a requirement that boxes be filtered out unless they form one-dimensional chains or two-dimensional rectangle where, for example, there should be at least three consecutive boxes in a straight line for them to be finally classified as foreground. The specific thresholds for this filter are a further workflow-parameter and a further detail that users might set interactively. With this final piece of information, the program can filter extraneous pseudo-foreground boxes and sum up the areas of the boxes that remain, yielding an approximate measure of the foreground's extent relative to the overall image.

This is a fairly coarse set of algorithms, which may or may not yield usable results when deployed against real-world images. However, even if more sophisticated methods (such as line- or keypoint-detectors) are ultimately employed instead, the interactive workflow just outlined could still be useful for preliminary experimentation with an image series. Similarly, the *evaluation* of the automated workflow described earlier can proceed via a series of image-transformations constructed via the interactive workflow. Whereas the latter workflow is designed to *calculate* the foreground-area by rotating and reducing the image to a two-color, coarse-boxed simplification, the evaluation performs similar steps so as to yield a binning classification (the foreground/background boxes becomes bins into which keypoints are sorted). As this example suggests, an interactive workflow can provide *assessment* of automated workflows even if the goal is for Computer Vision operations to run without human input. Such evaluations could help ensure the accuracy of automated pipelines and enable engineers to identify which image-

processing tactics and which workflow-parameters yield the best results for a given image-series.

For an image-processing environment to support interactive workflows, it must implement multiple GUI and event-handling requirements, typically ones that are not addressed by Computer Vision libraries themselves (such as OpenCV). Even the mathematically simplistic workflow reviewed here necessitates a number of special-purpose GUI controls, such as a color-picker, a tool for image-annotations (rhombus-shaped in particular, to get images' angular orientation), dialogs for setting thresholds and parameters on image-difference and criteria for filtering morphologically non-cohering foreground boxes, and controls for reporting pipeline/calculation results, graphically and/or numerically. Other GUI controls come into play when interactive workflows are adopted for evaluating automated pipelines (see Figure 4, especially 4a and 4b).

5 Conclusion

Interactive workflows are a prime example of how image-processing applications rely on GUI-level programming, but there are numerous other use-cases for which similar comments apply. For example, users might want to visually compare the results (or intermediate stages) of diverse workflows, spanning multiple techniques, to determine which approaches work best for a particular image-series. We can assume that in many cases parameters and algorithms which yield promising results on sample images within a series may be automatically applied to the image series, particularly if such images are thematically related (such that the series-elements have similar color patterns, textural complications, perspectival orientation — e.g. whether canonical background colors tend to arise from the top-center, as in outdoor/landscape pictures, or the left/right sides — etc.). Figure 6 shows a variety of GUI components associated with XCSD that demonstrate how applications can support users' experimenting with different algorithms (e.g., edge-detecting methods), visualizing color-schema, examining interactive color-histograms, and otherwise reviewing and fine-tuning data that will ultimately be incorporated into engineered workflows.

The more that applications seek to provide GUI support for image-processing operations, the more complex and expansive becomes the integration between Computer Vision libraries themselves and the GUI components which provide data and parameters to Computer Vision workflows. In this scenario one approach to engineering and maintaining the proper cross-component interoperability would be to support GUI controls, event-handlers, and data-exchange amongst the relevant software libraries via a Virtual Machine. This could entail VM code for designing GUI classes, during testing and development; VM intermediaries for handling events/signals generated by user actions through these GUI elements; VM-based protocols for connecting the GUI data to image-processing workflows implemented via isolated Computer Vision modules; or any combination of these.

Section 2 outlined specific image-processing workflows, enacted through the XCSD image format. That outline points to several details within an XCSD pipeline (arguably indicative of common concerns within many different Computer Vision workflows) which could involve cross-component interop, insofar as image-analysis is provided via dedicated plugins, extensions, or third-party libraries adopted by a host application. As mentioned above, a pipeline would often defer to the host at least for graphics-displays and initial image acquisition, but the XCSD workflow points to other moments at which interop may be necessary.

First, we mentioned the idea of consulting local histograms interactively so that human users might define foreground and background poles for sample/reference images in a series. In this case the host application would need to take responsibility for loading histogram displays and processing user actions. For "local" histograms the color counts are drawn from a small area — via XCSD the image is predivided into tierboxes anyhow, but in an interactive case users need to select one tierbox for which to view histogram data, which would be an action handled by a front-end component. Assuming XCSD code calculates the actual histogram and then sends this data back to the front-end, which in turn presents it visually, we see a multi-step exchange of data between the two components. Likewise, histogram displays may themselves be interactive — given that these visuals typically rank colors by showing bars filled with a particular color with a height documenting its occurrence in the histogram-area, the bars themselves can serve as color-selectors, so for example users could use histograms to select colors assigned specific thematic roles (such as foreground/background). Such actions would then generate further data routed back to the processing component (e.g., selected foreground/background poles applied to form a new one-channel reduction).

In the exchange just described, we assume that a front-end exposes capabilities to display histograms when provided the requisite data, while image-analysis components leverage this functionalities by assembling histogram data. Generalizing from this example, image-processing front-ends could support a variety of features relevant to initiating and examining Computer Vision pipelines, such as displaying images with annotations and overlays — this would include showing intermediate graphics wherein an image being analyzed is marked with icons, lines, or contours representing features extracted by detectors for Hough lines, scale/rotation invariant keypoints, boundary-contours, and similar algorithms. Host applications would thereby anticipate the general kinds of interactive/presentational features which are important to provide for an image-processing context, but they would remain open-ended with respect to the specific analytic modules that may target their front-end capabilities. In particular, host applications could then support a flexible spectrum of analytic methods, insofar as new sorts of algorithms could be introduced with their own mathematical constructions; each component can provide distinct analytic code so long as it accepts a common protocol for basic functionality involving acquiring, displaying, and invoking front-end features for analyzed images and image-series.

In short, the interop between image-processing host applications and analytic modules are a good example of bi-directional data sharing between semi-autonomous components, and fall under the rubric of interop situations where protocols may benefit from formalization or (at least partial) implementation via Virtual Machines.

As a concluding observation, readers may be interested in checking the chapter's demo code, which includes scripts (mostly build-tools) linked to over 100 cross-platform image-analysis libraries, offering a useful overview of the common practices and design patterns currently favored by Computer Vision developers. One noteworthy insight, reviewing the code for this collection of libraries, is that very few image-processing resources provide GUI tools or attempt to interoperate with GUI front-ends. Most of these repositories are focused on supplying processing algorithms (for segmentation, feature detectors, image-tagging, and so forth) and implement workflows with little connections to outside components, after acquiring an initial image to work on. Considering both the vast range of research that has been done on Computer Vision methods (evinced by the number of distinct

open-source tools, many of which codify methods presented through published research), and (by contrast) the paucity of application-integration tools, we might conclude that there is room for common standards and shared technologies for interoperating GUIs with image-processing modules to fill in a noticeable gap in the current Computer Vision ecosystem.

References

[1] Radhakrishna Achanta, *et. al.*, "SLIC Superpixels Compared to State-of-the-Art Superpixel Methods". https://www.cs.jhu.edu/~ayuille/JHUcourses/VisionAsBayesianInference2022/4/Achanta_SLIC_PAMI2012.pdf

[2] Martin Čadík, "Perceptual Evaluation of Color-to-Grayscale Image Conversions". http://cadik.posvete.cz/color_to_gray_evaluation/cadik08perceptualEvaluation.pdf

[3] A.C. Chadwick and R.W. Kentridge, "The perception of gloss: A review". https://pubmed.ncbi.nlm.nih.gov/25448119

[4] Darren M. Chan and Laurel D. Riek, "Object Proposal Algorithms in the Wild: Are they Generalizable to Robot Perception?". https://cseweb.ucsd.edu/~lriek/papers/chan-riek-IROS2019.pdf

[5] Jim X. Chen and Xusheng Wang, "Approximate Line Scan-Conversion and its Hardware Design". https://cs.gmu.edu/~jchen/report/hardwareLine.pdf

[6] Mingliang Chen, *et. al.*, "Interactive Hierarchical Object Proposals". http://www.shengfenghe.com/qfy-content/uploads/2019/12/458b6d07423a8844d3129412b717a0b3.pdf

[7] Ming-Ming Cheng, *et. al.*, "Deep Hough Transform for Semantic Line Detection". https://arxiv.org/pdf/2003.04676.pdf

[8] Gideon Kanji Damaryam, "A Method to Determine End-Points ofStraight Lines Detected Using the Hough Transform". https://www.ijera.com/papers/Vol6_issue1/Part%20-%201/H601016775.pdf

[9] Matt Duckham, *et. al.*, "Efficient Generation of Simple Polygons for Characterizing the Shape of a Set of Points in the Plane". http://geosensor.net/papers/duckham08.PR.pdf

[10] Mark Ewald, "Content-Based Image Indexing and Retrieval in an Image Database for Technical Domains". http://www.ibai-publishing.org/journal/issue_mldm/2009_july/mldm_2_1_3-22.pdf

[11] Davit Gigilashvili, *et. al.*, "Perceived Glossiness: Beyond Surface Properties". https://jbthomas.org/Conferences/2019aCIC.pdf

[12] Xiaoying Guo, *et. al.*, "Analysis of Texture Characteristics Associated with Visual Complexity Perception". https://link.springer.com/article/10.1007/s10043-012-0047-1

[13] Bumsub Ham, *et. al.*, "Proposal Flow: Semantic Correspondences from Object Proposals". https://www.di.ens.fr/willow/pdfscurrent/ham2017.pdf

[14] Kunimatsu Hashimoto, *et. al.*, "KOSNet: A Unified Keypoint, Orientation and Scale Network for Probabilistic 6D Pose Estimation". http://groups.csail.mit.edu/robotics-center/public_papers/Hashimoto20.pdf

[15] Zheyuan Hu, *et. al.*, "Object Pose Estimation for Robotic Grasping based on Multi-view Keypoint Detection". http://www.cloud-conf.net/ispa2021/proc/pdfs/ISPA-BDCloud-SocialCom-SustainCom2021-3mkuIWCJVSdKJpBYM7KEKW/264600b295/264600b295.pdf

[16] Csaba István Kiss, *et. al.*, "On Feature Extraction for Texture Analysis". https://core.ac.uk/download/pdf/236628682.pdf

[17] Sifan Ji, *et. al.*, "Image Clustering Algorithm using Superpixel Segmentation and Non-Symmetric Gaussian-Cauchy Mixture Model". https://ietresearch.onlinelibrary.wiley.com/doi/pdf/10.1049/iet-ipr.2020.0402

[18] Zequn Jie, *et. al.*, "Scale-Aware Pixelwise Object Proposal Networks". https://www.sysu-hcp.net/userfiles/files/2021/02/28/3a62f36bc1fe7fb9.pdf

[19] V. R. Jyothish, *et. al.*, "An Efficient Image Segmentation Approach using Superpixels with Colorization". https://www.sciencedirect.com/science/article/pii/S1877050920310620

[20] Philipp Krähenbühl and Vladlen Koltun, "Geodesic Object Proposals". http://vladlen.info/papers/gop.pdf

[21] Giovane R. Kuhn, *et. al.*, "An Improved Contrast Enhancing Approach for Color-to-Grayscale Mappings". https://www.inf.ufrgs.br/~oliveira/pubs_files/MS_C2G/Kuhn_Oliveira_Color_to_Gray_TVCJ_Pre-print.pdf

[22] Rushi Lan, *et. al.*, "Quaternionic Local Ranking Binary Pattern: A Local Descriptor of Color Images". https://ieeexplore.ieee.org/document/7350143

[23] Stefan Leutenegger, *et. al.*, "BRISK: Binary Robust Invariant Scalable Keypoints". https://margaritachli.com/papers/ICCV2011paper.pdf

[24] Nannan Liao, *et. al.*, "BACA: Superpixel Segmentation with Boundary Awareness and Content Adaptation". https://www.mdpi.com/2072-4292/14/18/4572

[25] Maiko M. I. Lie, *et. al.*, "Joint Upsampling of Random Color Distance Maps for Fast Salient Region Detection". https://www.sciencedirect.com/science/article/abs/pii/S0167865517303215

[26] Wei Hong Lim and Nor Ashidi Mat Isa, "Color to Grayscale Conversion Based On Neighborhood Pixels Effect Approach for Digital Image". https://www.emo.org.tr/ekler/14281fad9e3d0cc_ek.pdf

[27] Xiu Liu and Chris Aldrich, "Deep Learning Approaches to Image Texture Analysis in Material Processing". https://mdpi-res.com/d_attachment/metals/metals-12-00355/article_deploy/metals-12-00355-v3.pdf?version=1645497882

[28] Yao Lu and Linda Shapiro, "Closing the Loop for Edge Detection and Object Proposals". https://homes.cs.washington.edu/~shapiro/yao-aaai.pdf

[29] Subhransu Maji and Jitendra Malik, "Object Detection using a Max-Margin Hough Transform". https://www2.eecs.berkeley.edu/Research/Projects/CS/vision/shape/smaji-cvpr09.pdf

[30] Fereshteh Mirjalili and Jon Yngve Hardeberg, "On the Quantification of Visual Texture Complexity". https://www.mdpi.com/2313-433X/8/9/248

[31] László Neumann, *et. al.*, "An Efficient Perception-based Adaptive Color to Gray Transformation". https://diglib.eg.org/bitstream/handle/10.2312/COMPAESTH.COMPAESTH07.073-080/073-080.pdf?sequence=1

[32] Chuong T. Nguyen and Joseph P. Havlicek "Color to Grayscale Image Conversion using Modulation Domain Quadratic Programming". https://ieeexplore.ieee.org/document/7351674

[33] Shin'ya Nishida "Image Statistics for Material Perception". https://www.sciencedirect.com/science/article/pii/S235215461930035X

[34] Ivan Papusha and Matthew Ho, "Hough Transform for Directional Orientation". https://stacks.stanford.edu/file/druid:rz261ds9725/Ho_Papusha_RealTimeHoughTransform.pdf

[35] Jordi Pont-Tuset, *et. al.*, "Multiscale Combinatorial Grouping for Image Segmentation and Object Proposal Generation". https://upcommons.upc.edu/bitstream/handle/2117/105287/1503.00848.pdf

[36] Antonio Rubio, *et. al.*, "BASS: Boundary-Aware Superpixel Segmentation". https://digital.csic.es/bitstream/10261/166227/1/bassbound.pdf

[37] Giorgos Sfikas, *et. al.*, "Quaternion Harris For Multispectral Keypoint Detection". https://www.cs.uoi.gr/~sfikas/icip-quatharris.pdf

[38] Hadi Sutopo, "Bresenham's Lines Algorithm Visualization Using Flash". https://ggn.dronacharya.info/ITDept/Downloads/QuestionBank/Odd/V%20sem/Bresenham_Lines_Algorithm_Visualization_Flash_IIISem.pdf

[39] Mihran Tuceryan and Anil K. Jain, "Texture Analysis". https://cs.iupui.edu/~tuceryan/research/ComputerVision/texture-review.pdf

[40] Takeshi Uejima, *et. al.*, "Proto-Object Based Saliency Model With Texture Detection Channel". https://www.frontiersin.org/articles/10.3389/fncom.2020.541581/pdf

[41] Esteban Uriza, *et. al.*, "Efficient Large-scale Image Search with a Vocabulary Tree". https://www.ipol.im/pub/art/2018/199/article.pdf

[42] Xinming Wang, "Real-time Classified Hough Transform Line Detection Based on FPGA". https://www.atlantis-press.com/article/25848213.pdf

[43] Zhi-Zhong Wang and Jun-Hai Yong, "Texture Analysis and Classification With Linear Regression Model Based on Wavelet Transform". https://hal.inria.fr/inria-00517305/document

[44] Gunnar Wendt and Franz Faul, "Factors Influencing the Detection of Spatially-Varying Surface Gloss". https://journals.sagepub.com/doi/pdf/10.1177/2041669519866843

[45] Jianping Wu, *et. al.*, "Real-time Vanishing Point Detector Integrating Under-parameterized RANSAC and Hough Transform". https://openaccess.thecvf.com/content/ICCV2021/papers/Wu_Real-Time_Vanishing_Point_Detector_Integrating_Under-Parameterized_RANSAC_and_Hough_Transform_ICCV_2021_paper.pdf

[46] Minglong Xue, "Deep invariant texture features for water image classifcation". https://link.springer.com/content/pdf/10.1007/s42452-020-03882-w.pdf

[47] Yi-Xuan Zhan, *et. al.*, "Superpixel Segmentation Using Improved Lazy Random Walk Framework Based on Texture Complexities". http://www.csroc.org.tw/journal/JOC30_5/JOC-3005-20.pdf

[48] Ziming Zhang and Philip H.S. Torr, "Object Proposal Generation using Two-Stage Cascade SVMs". https://zimingzhang.files.wordpress.com/2014/10/ziming_tpami_obj_proposal_final.pdf

[49] Hanli Zhao, *et. al.*, "Image Recoloring using Geodesic Distance Based Color Harmonization". https://core.ac.uk/download/pdf/81904494.pdf

[50] Shan Zhao and Jing Liu, "Image Descriptor Based on Local Color Directional Quaternionic Pattern". https://www.spiedigitallibrary.org/journalArticle/Download?urlId=10.1117%2F1.JEI.28.4.043003

[51] Yu Zhou, *et. al.*, "Object-level Proposals". https://openaccess.thecvf.com/content_ICCV_2017/papers/Ma_Object-Level_Proposals_ICCV_2017_paper.pdf

[52] C. Lawrence Zitnick and Piotr Dollár "Edge Boxes: Locating Object Proposals from Edges". https://link.springer.com/content/pdf/10.1007/978-3-319-10602-1_26.pdf

[53] Jana Zujovic, "Structural Texture Similarity Metrics for Image Analysis and Retrieval". http://users.eecs.northwestern.edu/~pappas/papers/zujovic_tip13.pdf