This is a repository copy of *A software system for laboratory experiments in image processing*.

White Rose Research Online URL for this paper:
http://eprints.whiterose.ac.uk/1135/

**Article:**

https://doi.org/10.1109/13.883358

**Reuse**
Items deposited in White Rose Research Online are protected by copyright, with all rights reserved unless indicated otherwise. They may be downloaded and/or printed for private study, or other acts as permitted by national copyright laws. The publisher or other rights holders may allow further reproduction and re-use of the full text version. This is indicated by the licence information on the White Rose Research Online record for the item.

**Takedown**
If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.
A Software System for Laboratory Experiments in Image Processing

John A. Robinson, Member, IEEE

Abstract—Laboratory experiments for image processing courses are usually software implementations of processing algorithms, but students of image processing come from diverse backgrounds with widely differing software experience. To avoid learning overhead, the software system should be easy to learn and use, even for those with no exposure to mathematical programming languages or object-oriented programming. The class library for image processing (CLIP) supports users with knowledge of C, by providing three C++ types with small public interfaces, including natural and efficient operator overloading. CLIP programs are compact and fast. Experience in using the system in undergraduate and graduate teaching indicates that it supports subject matter learning with little distraction from language/system learning.

Index Terms—C++ class libraries, image processing education, image processing software.

I. INTRODUCTION

MANGE processing and analysis figure in many electrical engineering and computer science curricula, usually in elective courses at the senior level. Increasingly they are also of interest to students in mechanical engineering (for their application to robotics), geography (remote sensing) and physiology (medical imaging and ophthalmology). Assuming such students fulfill the mathematical prerequisites for an image processing course (in multivariate analysis, signal processing, etc.), they will enter it with diverse software backgrounds. Some students will have taken only one programming course at the university level. Laboratory experiments are an important pedagogical tool, as well as preparation for practice in the field, but in image processing they almost always involve software. The problem is to provide these experiments within an environment that requires as little “overhead” learning as possible, even for inexperienced programmers. At the same time, the environment should provide a route toward high-quality program production, supporting research and design projects (which may be components of an advanced course).

The requirements for such a software system can be captured in four goals:

1) It should be easy to learn how to use the system, starting from the users’ prior knowledge.
2) Simple image processing tasks should be implemented in small programs. Difficult tasks should be enabled by compact combination of simple subtasks.
3) Programs should be as fast as possible.
4) Real-time image output should be possible and simple.

That is, it should be possible to see the progress of an image processing program as it progressively generates its output. Similarly, real-time image input should be possible and simple, so that users can apply their programs to live video.

The first two goals address the diversity of prior programming experience and the need to gain facility with the environment with little overhead; the third and fourth reflect the need to use the system on real projects. The fourth goal also provides for immediate feedback during the run of an algorithm, speeding up design and experimentation.

The solution proposed in this paper is a class library for image processing (CLIP). This software system meets the four goals and has been used in teaching image processing for several years. Section II of the paper reviews prior technologies that meet some of the goals, and examines students’ familiarity with those technologies. Section III summarizes the major features of CLIP as seen by the user, giving examples of programs. Some implementation details are discussed in Section IV. Section V reports experience in using the system in undergraduate and graduate image processing courses, and Section VI concludes with comments on future work.

II. BACKGROUND

CLIP’s development was inspired by four strands of prior art. Each of these partially fulfills the goals stated in Section I.

1) The C Programming Language [1].: C is the image processing field’s most widely used programming language. It is small and efficient, and well-matched to the procedural nature of image manipulation. Many libraries of C functions have been written to support image processing. More significant pedagogically is that C is closer to being universal than any other computer language. Many computer science and computer engineering programs now favor Java for introductory programming courses, while other disciplines use C++. Fortran, Matlab, or Visual Basic. Despite this, C’s importance is threefold: it is a parent of C++, Java, and Matlab, and therefore easily understood from the perspective of these larger languages, it has even more legacy code than Fortran, and as a subset of C++, it is dominant in many industry applications. Of course, C has well-known problems, but its prominence means it is a natural choice for fulfilling the goal of easy learning. Although the teacher of image processing can mandate use of an object-oriented programming language, the extra learning time involved is not necessarily well spent. Although CLIP uses C++, for
reasons explained below, it was designed to require minimal
learning by a student already knowledgeable in C, and to en-
courage procedural programming in the C style.

2) *Matrix Programming Languages.*: Matrix programming
languages (MPLs) such as Matlab [2], provide concise syntax
for matrix operations. By expressing images as matrices, many
fundamental image processing operations can be accomplished
simply, though not necessarily efficiently. Thus it is possible to op-
erate on subimages (submatrices) with arithmetic operations that
are applied pointwise or vectorially. Moreover, Matlab is ideal for
linear algebraic calculations, such as the eigenvalue analysis used
in feature extraction or Weiner filtering. With its excellent signal
and image processing toolkits, Matlab can be a very powerful fast
prototyping tool in image processing and analysis.

Although MPLs, and Matlab in particular, meet goal 2) above,
they do not meet the other goals. Many students do not have prior
MPL experience, and while learning Matlab may be a benefit that
transcends a single course, the time involved is significant. MPL
code can be inefficient. It is not uncommon to replace an MPL
prototype with a production version written in a procedural lan-
guage. Matlab includes a translator to C/C++ which increases the
execution speed for operations involving scalars. However the
resulting code is still significantly slower than well-written C/C++
for fixed-size images. Finally, the real-time input–output faciliti-
ties of MPLs are neither transparent to the coder nor as efficient
as C/C++ alternatives.

3) *The C++ Programming Language* [3]: C++ is a
portable, efficient, extensible object-oriented programming
language. It allows the creation of data types which can be used
with the same syntactic economy as the matrices of Matlab,
etc. The pel-by-pel and block-by-block iterations required by
many algorithms can be conveniently handled within suitable
objects. Range limiting can be implemented as a modification of
conventional indexing. Error handling can be devolved to generic
types. Furthermore, users of a class can extend it freely, whereas
MPL users have limited ability to extend the matrix type.

There are several examples of class libraries which aim to unite
the programming benefits of C/C++ and matrix programming
languages. These range from the powerful general-purpose (e.g.,
M++ [4]), to focused developments for particular applications
(e.g., embedded signal processors [5]). Such systems are designed
with the expectation that users are already familiar with C++.
Unfortunately C++ is a big language, and to appreciate its full
power requires considerable learning beyond C.

One criticism of both matrix programming languages and
C++ matrix class libraries is that they are too general for image
processing. The reason is that dynamic (resizable) arrays are the
data type at the heart of such systems, but images are rarely re-
sized, and the overhead of this flexibility is rarely warranted.

CLIP therefore introduces a small set of C++ data types. Stu-
dents can use these with little learning beyond C; they provide
for very tight coding in the style of Matlab, but without the over-
heads of dynamic resizing; they incorporate interfaces to pic-
tures and input–output that allow the user to access stored and
live images and subimages with simple array-like operations.

4) *The Partitioned Image Processing System.* The partition-
toned image processing system (PIPS) was developed by the
author in 1983 for use on custom-designed hardware which
included memory-mapped image frame stores [6]. While the
architecture constrained programs to use limited-precision
integer arithmetic, this disadvantage was offset by the speed
and real-time visual output that the system provided. Perhaps
the main stimulus to the development of CLIP was the (sur-
prising) realization that many learners and researchers in image
processing do not have the benefit of real-time visual feedback
during processing, which is so useful in program development.
Instead they must suffer systems where processing and display
are done in two separate stages. This is true of almost all C
and C++ image processing libraries, as well as Matlab. Thus,
real-time image output was an important requirement for CLIP.
When there are no constraints on a processed picture’s dynamic
range, the system must monitor that range to adapt the display
level mapping as the processing proceeds.

III. CLIP DESIGN

A. Overview

CLIP was designed to unite salient facilities of the four
strands of prior art described in Section II to achieve the four
goals in Section I. It is a class library in C++, but its users’
required knowledge of C++ is elementary, corresponding
roughly to the first quarter of a typical textbook (such as [7]). It
therefore supports fast migration from C. There are only three
new data types, and the class library’s public interface is small.
Learning is fast, and is mainly done through example programs
that illustrate compact programming with CLIP.

Reading and writing of images, error detection, and warnings
are all provided in the class library, keeping user programs short.
The library types provide natural iterations over arithmetic op-
erators and callbacks, but there are no separate iterator types
which could confuse novices.

CLIP allows real-time visual feedback during processing.
In its current Unix/X-Windows version, image viewing can be
turned on or off for any picture object. Iteration operations can
be made to update an onscreen window after each processed
row or column of the image.

B. CLIP Data Types—Picture, Irange, and Vrange

CLIP augments the built-in data types with just three addi-
tions: the picture, the integer range (irange) and the value
range (vrange). Associated with these are overloaded oper-
ators that support arithmetic on and between the types. The
picture type stores a two-dimensional array of pel values and
provides member functions and overloaded operators for its ma-
nipulation. Picture objects can be added, subtracted, assigned,
etc. using the conventional assignment operators (+=, -=, =,
etc.), and can similarly be combined with built-in types such as
int and float. An object of the irange type represents a sequence
of integers such as

\[ 0, 1 \cdots 7 \text{ (integers from 0–7), or } 0, 3 \cdots \text{pic}_\text{size} \text{ (every}
\text{third integer up to pic}_\text{size}, \text{while a vrange object rep-}
\text{resents a continuous range of values, such as <128, or [10,}
\text{20]}. Arithmetic on iranges and vranges is supported,
yielding combined or shifted ranges.
#include "usage.h"
#include "picture.h"

void lap(int pel, const int **buf)
{
    pel = 4*buf[0][0]-buf[-1][0]-buf[0][-1]-buf[0][1]-buf[1][0];
}

void main(int argc, char *argv[])
{
    usage("laplacian inpic outpic"); // (usage is a macro defined in usage.h that
    // counts the number of words in the string,
    // and if this is not equal to argc, prints
    // a usage message and exits.)

    picture inpic(argv[1]);  // Make inpic from data in file
    picture outpic(in, lap);  // Make outpic by iterating lap over inpic
    outpic.write(argv[2]);   // Write outpic to file
}

Fig. 1. A complete CLIP program for a simple Laplacian approximation, illustrating the use of a callback.

It is appropriate to state now a simple rule about arithmetic on picture objects that CLIP imposes: the result of picture +
picture, or picture + int (or picture + float—see Section V), where * is any of the binary arithmetic operators (+, -, /, etc.), is an integer (or float) which is the sum of the results of the operation applied pointwise over the picture. So the combination picture + picture yields not a picture, but a number. To get a picture as the result of a pointwise arithmetic operation, the appropriate assignment operator must be used, for example, picture += picture. The reason for this constraint is given in Section IV below. The rule applies only to pictures, not to irranges and vranges.

The subscripting operator [ ] is overloaded to allow pictures to be indexed by irranges and vranges. Consequently a picture block can be defined compactly as in this program fragment

    picture in('test');
    //Create a picture object called in and
    //load in data from a file called test
    picture block(blockrows, blockcols);  //To represent the block of pels
    irrange rowrange (0, 1, blockrows - 1);
    irrange colrange (0, 1, blockcols - 1);
    block = in[rowrange + startrow][colrange + startcol];
    //(startrow and startcol are integers)

Similarly, a noncontiguous irrange can be used to downsample, and vrange subscripting used to generate masks. The compact combination of two types and arithmetic operators is illustrated in the following statement, taken from a one-page program that does motion-compensated interframe prediction using full-search block matching.

    matches [ti][tj] = curr[yblock+i][xblock+j] \^ prev[yblock+i+ti][xblock+j+ti].

The object types on the left-hand-side are picture[int]int: this returns a reference to an integer (or a float—see below) which is the pel value at location (ti, tj) in the picture matches. The object types on the right-hand side are

    picture[irange + int][irange + int] \^ picture[irange + int + int][irange + int + int].

The ^ operator performs a pointwise squared difference operation on the two picture blocks defined by the picture objects subscripted by irranges. The result is an int, the total squared difference between the two blocks. Thus the value of matches (ti, tj) is the difference between the block in the current frame and an equivalent block in the previous frame displaced by (ti, tj).

C. Callbacks

CLIP supports the use of callbacks for point, neighborhood and block operations. The program given in Fig. 1 illustrates this. In this program, the constructor for outpic works by iterating a callback function (lap) on inpic. The callback, lap, calculates a simple Laplacian for the current point. In this example, as for all neighborhood callbacks, the arguments pel and buf refer to the destination and source of the neighborhood operation. pel and buf are set up on each call so that buf[0][0] has the same (row, col) position in the source picture as pel has in the output. Point, neighborhood and block operations are supported explicitly on multiple picture objects as well as in constructors.

D. CLIP In Use

Fig. 2 shows a complete vector quantization coder implemented in CLIP. This example is typical of the structure and length of CLIP programs used for standard image processing tasks. Block subscripting has been used to implement the innermost iteration loops, but outer loops are done conventionally. Multiple levels of iteration were designed in to the first version of CLIP, to support, for example, convolution of two range-limited picture's in a single statement. However, users find the interpretation (and writing) of statements with multiple irranges to be difficult and error-prone. Hence the mixture of loops and range subscripting is most common.
#include "usage.h"
#include "picture.h"

// Represents inpic using blocks taken from a codebook file and writes result to
// outpic. The codebook is stored as a 2D array of blocks (any aspect ratio), so
// that it can be manipulated and displayed as a picture. The indices of the
// selected vectors (blocks) are written (in ASCII) to stdout. Writes to stderr
// the mean square error of the vq representation

const int blocksize = 4;
const char *codefile = "codebook.4x4";

void main(int argc, char *argv[]) {
    usage("vq inpic outpic");
    picture inpic(argv[1]);
    picture code(codefile);
    picture outpic(inpic.nrows(), inpic.ncols());
    range block(0,1,blocksize-1);
    picture matches(inpic.nrows() / blocksize, code.ncols() / blocksize);
    for (int i = 0; i < inpic.nrows(); i += blocksize) {
        for (int j = 0; j < inpic.ncols(); j += blocksize) {
            // For every block in the input picture
            for (int ti = 0; ti < code.nrows(); ti += blocksize) {
                for (int tj = 0; tj < code.ncols(); tj += blocksize) {
                    // For every block in the codebook
                    matches[ti/blocksize][tj/blocksize] =
                        inpic[block+i][block+j] ^ code[block+ti][block+tj];
                }
            }
        }
    }

    int besti, bestj;
    int best = matches.min(besti, bestj);
    outpic[block+i][block+j] =
        code[block+(besti*blocksize)][block+(bestj*blocksize)];
    cout << besti*code.ncols() / blocksize + bestj;
}

int mse = (outpic ^ inpic) / (inpic.nrows() * inpic.ncols());
cerr << "Mean square error = " << mse << 'n';
outpic.write(argv[2]);
}

Fig. 2. A complete CLIP program for vector quantization coding.

Figs. 1 and 2 are examples of a large range of programs
written under CLIP, most for practical experimentation. CLIP
has been used for research as well as teaching. References [8]
and [9], as well as ongoing projects in edge detection, motion
estimation and coding, make use of CLIP.

E. Handling of Picture Element Types

An unresolved design issue in CLIP is how pictures of dif-
cerent element types should be handled. The library at present
exists in two forms. The first has picture as a template. Picture
elements may be integers, floats, complex numbers, etc. as
specified by the user, via the normal C++ template mechanism.
The second library contains two data types, picture_of_int
and picture_of_float, with the obvious meanings. While the
first version is the more flexible, and certainly more in the spirit
of C++, the second may be preferable in practice. Intended
users of the class library are new to type abstraction, and may
balk at the kind of second-order abstraction of templates. For
this reason only the second version of the library has been used
in teaching.

IV. IMPLEMENTATION ISSUES

CLIP avoids one major problem of flexible matrix class li-
braries by its implementation of binary arithmetic operations
between pictures as numerical sums. If picture + picture
yielded not a number, but a picture, it would be necessary
to manage the creation and destruction of arbitrary numbers of
temporary pictures resulting from arithmetic expressions like

    picture = (picture + picture
              - (picture * (picture/picture))).

But because arithmetic generating pictures from pictures is only
implemented in the assignment operators ( +=, -=, etc.), there
is no need for temporary pictures. This, together with the fact
that pictures are not resizeable, means that CLIP's memory man-
agement tasks are well defined and therefore fast.
Most of the complexity in the class library is to do with efficient dereferencing of pictures. For example, supporting `picture[int][int]range` necessarily means that `picture[int][int]` will be slower than normal indexing, but the overhead must be very small. Otherwise C programmers, who are likely to use `picture[int][int]` heavily, will see the class library as ponderous. Furthermore, the use of callbacks, as encouraged by programs such as the example Laplacian given in Fig. 1, must be efficiently supported by ordering iterations to minimize the number of counter and pointer increments. As an example of the strategies taken to optimize efficiency, the implementation of callbacks is discussed now.

CLIP uses a fairly standard implementation of a two-dimensional matrix as a contiguous one-dimensional (1-D) array plus an array of "row pointers" into the start points of each matrix row. The main 1-D array includes padding pseudopels around the image data, but this is hidden from the user who accesses the data only via the row pointers. Callbacks are implemented to process the picture columnwise. For each pel in the first column a pointer to the row pointer to the pel in question is passed to the callback function. Once the entire column has been processed (from top to bottom), all the row pointers are incremented and the next column is handled in a similar way. When all the columns have been done, the row pointers are all reset to point to the beginning of their respective rows. The advantage of this approach is that the callback can use pels at any offset from (what it sees as) `buf[0][0]`.

The callback strategy in CLIP means that very simple programs, like the Laplacian in Fig. 1, take up to three times longer to run than well-written conventional (single function) C versions. For more complicated programs this difference goes down rapidly; the per-pel processing common to both cases takes longer, but the callback overhead is fixed.

On the other hand, programs involving range-indexed picture blocks are sometimes faster in the CLIP version than in a C equivalent. This is because the iterative loops within the `picture` data type have been highly tuned. As an example, the vector quantizer shown in Fig. 2 is slightly faster than its C equivalent for at least one optimizing compiler.

V. USAGE EXPERIENCE

CLIP was trialed in an undergraduate image processing course, then subsequently used in undergraduate and graduate courses in image communications. Each class consisted of students from diverse backgrounds. Informal surveys indicated that all had prior knowledge of the C programming language, most had prior C++ experience and many were familiar with Java and Matlab. There was a very strong student preference for C/C++.

In the image communications courses, students did four lab assignments using the software system. The labs were done singly without instructor or TA assistance. The subjects were:

- two-level image manipulation and lossless compression;
- predictive coding and errors;
- transform, Laplacian pyramid, and wavelet coding;
- motion estimation.

In each case the students were provided with sample programs to use in experiments, then expected to write their own programs according to some problem statement. At the end of the course, students completed feedback forms, including several questions on the effectiveness of the lab experiments using CLIP.

All students understood and effectively used the CLIP classes in all four labs. Errors were usually in procedural parts of the code, unaffected by the object-orientation. There was no misunderstanding or misuse of overloaded operators. Overall, students performed well. Their feedback to the labs was overwhelmingly positive, and they explicitly confirmed that in the work required for successful completion, language and class library learning were insignificant.

There was no measurable difference in student performance or feedback between former C++ users and nonusers.

VI. CONCLUSION

CLIP, a class library for image processing in C++, has been designed, and successfully used, to support image processing education. It incorporates matrix-like operations into C++ in a framework which makes very few learning demands on C programmers.

For those learning image processing, interactivity and strong feedback are important cues. It is possible within CLIP to create picture objects that are shown on-screen during processing. Further investigation is required to assess the value of this immediate feedback on learning and algorithm development.

REFERENCES


John A. Robinson (M’87) received the B.Sc. degree from Durham University, U.K., in 1979 and the M.Sc. and Ph.D. degrees from the University of Essex, U.K., in 1982 and 1985, respectively.

He was a Development Engineer with Standard Telephones and Cables Ltd., Basildon, U.K., and a Member of Scientific Staff, then Manager, Multimedia Conferencing, with Bell-Northern Research Ltd., Verdun, PQ, Canada. He was with the University of Waterloo, ON, Canada, from 1988 to 1995, including a sabbatical year spent at the Universities of Essex and Cambridge, U.K. From 1996 to 2000, he was the NSERC/NetAd Ltd Industrial Research Chair in Telecommunications Engineering and Information Technology at Memorial University of Newfoundland. There he established and led the Multimedia Communications Laboratory. In October 2000, he became Professor of Electronics at the University of York, U.K.