Media ECAD. CLIP tutorial 3.

Blocks, meshes and geometric transformations

John Robinson

3.1 The `irange` class

CLIP only has three data types or classes[1]. `colour_picture` and `picture_of_int` were introduced in the first week of this tutorial. The third and last data type, `irange`, is usually used in combination with either `colour_picture` or `picture_of_int` to process many pels simultaneously.

An `irange` object represents a sequence of equally-spaced integers such as (1, 2, 3) or (3, 7, 11, 15, 19) or (0, 1, 2, 3, … 511). It is instantiated through a statement like

```
irange block(0,1,7);
```

which initializes block to represent the sequence (0,1,2,3,4,5,6,7). In general the three arguments in the constructor are the first, second and last integers in the sequence. So

```
irange evenpoints(0,2,254);
irange oddpoints(1,3,255);
```

set up two sequences consisting respectively of all the even and all the odd integers from 0 to 255.

Other ways to construct an irange are:

```
irange justonepoint(67);       // Sequence with just one value
                               // (equivalent to (67, 67, 67)).
irange anotherblock(block);    // Copy constructor
```

You can do simple arithmetic reassignment on an irange such as

```
evenpoints += 64;
```

which, with `evenpoints` as constructed earlier, would change it to the sequence (64, 66, 68, 70, … 318). It is also possible to use expressions like `(200 + block)` which evaluates to (200,201,202,203,204,205,206,207) for the `irange` object `block`, but doesn't change `block`'s value.

An `irange` object is used to index a `colour_picture` or a `picture_of_int` to process a block of pels simultaneously. This idea is borrowed from Matlab, and, as in Matlab, it is a very powerful feature, but takes a little while to get used to. Overleaf is a program that demonstrates some of the many things you can do by indexing `colour_picture`s or `picture_of_int`s with `irange`s. Although the program's only purpose is to show how `irange` works, it is worth studying carefully and implementing. (This is the last long program that you will be asked to enter by hand from the tutorial, but please don't skip the entering, compiling, debugging, testing, and examining of the output, because a good understanding of this

[1] Earlier versions included classes like `picture_of_double`, `vrange`, and `picture` as a template class. These are mentioned in the IEEE Trans on Education paper, but by the time it was published I had already eliminated them to make the library simpler.

code will equip you to use `iranges` and picture arithmetic.) The code contains several comments which should guide you through its processing of 16 square blocks. Ensure you understand how the code moves from block to block. Notice that the input from the other picture is not always from the same block as the main picture. Also note that there are three assignment operators on pictures that do not have their conventional meanings:

`destpic |= sourcepic;`	replaces each pel in `destpic` (or in the `irange` indexed block of `destpic`) with the absolute difference between that pel and the corresponding pel in `sourcepic`.
`destpic ^= sourcepic;`	replaces each pel in `destpic` with the squared difference between that pel and the corresponding pel in `sourcepic`.
`destpic &= sourcepic;`	does conditional replacement. For each pel in `destpic`, if the corresponding pel in `sourcepic` is non-negative, the value is replaced by the `sourcepic` value. Otherwise the `destpic` value is unaffected. This provides a simple kind of transparency that can be used for processing irregularly-shaped sprites or picture layers.

```
// Program to demonstrate some of the features of irange
#include "picture.h"
// Some callbacks used in the demonstration
int thresh1(int& pel) {
       if (pel < 128) pel = 0;
       else pel = 255;
       }
int thresh2(int& outpel, const int& inpel) {
       if (inpel < 128) outpel = 0;
       else outpel = 255;
       }
int average(int& outpel, const int **buf) {
       outpel = buf[-1][-1] + buf[-1][0] + buf[-1][1] +
              buf[0][-1]  + buf[0][0]  + buf[0][1] +
              buf[1][-1]  + buf[1][0]  + buf[1][1];
       outpel /= 9;
       }

int main(int argc, char *argv[]) {
       if (argc != 1) {
              cerr << "Usage: irangedemo";
              exit(1);
              }
       colour_picture pic("../pics/baboon512");
       colour_picture other("../pics/lenna512");
       irange basic(0,1,127);     // Basic block size
       irange evensamples(0,2,254);      // For demos with subsampling
       irange rowrange(basic);
       irange colrange(basic);
       irange otherrowrange(basic);
       irange othercolrange(basic);
       // First row of four blocks demos operations with integers
       // Leave very first block alone: Don't do anything

       // For second block, set whole block equal to 50
       colrange += 128;
       pic[rowrange][colrange] = 50;
       // Invert third block
       colrange += 128;
       pic[rowrange][colrange] *= -1;
       pic[rowrange][colrange] += 256;   // Move back into [0,255] range
       // Halve amplitude of fourth block
       colrange += 128;
```

```
pic[rowrange][colrange] /= 2;

// Second row of four blocks demos arithmetic operations with
// another picture
// First block is just copy from (top left block of) other picture
colrange = basic;   // Reset colrange to (0,1,127)
rowrange += 128;
pic[rowrange][colrange] = other[otherrowrange][othercolrange];
// Second block is copy from elsewhere in other picture
colrange += 128;
pic[rowrange][colrange] = other[200+otherrowrange][200+othercolrange];
// (Note how adding an int to an irange yields a shifted irange)

// Third block is average of block at this location in two pictures
colrange += 128;
pic[rowrange][colrange] += other[rowrange][colrange];
pic[rowrange][colrange] /= 2;
// Fourth block is difference between two blocks (plus an offset)
colrange += 128;
pic[rowrange][colrange] -= other[rowrange][colrange];
pic[rowrange][colrange] += 128; // To give vals mostly in [0,255].

// Third row of four blocks demos the three non-arithmetic operators
// and iranges with different spacings.
colrange = basic;
rowrange += 128;
// The operator |= produces the absolute difference between blocks
pic[rowrange][colrange] |= other[rowrange][colrange];

// The operator ^= produces the squared difference between blocks
colrange += 128;
pic[rowrange][colrange] ^= other[rowrange][colrange];
// Have to scale result or else whole picture will display poorly
int blockmax = pic[rowrange][colrange].max(); // NOTE: max() applies
                                              // just to specified
                                              // block.
pic[rowrange][colrange] *= 255;
pic[rowrange][colrange] /= blockmax;

// The operator &= does conditional replacement. If the pel in
// the source picture is greater than or equal to 0, it replaces
// the destination picture. If less than 0, the source pel
// remains.
// To illustrate this, we need to create a masked version of part
// of the other picture
colour_picture temppic(128,128);
temppic = other[rowrange][colrange];
// Let's just preserve a disk of temppic, by putting negative values
// into the pels outside a disk.
const int radius = 50;
for (int i = 0; i < 128; i++)
for (int j = 0; j < 128; j++) {
       int x = j - 64;     // Relative to centre of pic
       int y = i - 64;
       if (x*x + y*y > radius*radius)
              temppic[i][j] = -1;
       }
// Now, &= should do appropriate masking
colrange += 128;
pic[rowrange][colrange] &= temppic;

// Using the fourth block on the third row, we demonstrate
// subsampling a source
```

```
        // block and putting it in a destination block

        colrange += 128;
        pic[rowrange][colrange] = other[evensamples][evensamples];
        // Note that although this is the only example where the other
        // samples have a different spacing, all the operators illustrated
        // here work no matter what the irange spacings. What counts is
        // that the number of integers in the irange sequences are the same.


        // The final row of four blocks demos member functions applied just
        // to irange-specified blocks.
        // (Already seen max() member function on a block.
        // min() works similarly.)

        // Write a block of the picture to a file
        colrange = basic;
        rowrange += 128;
        pic[rowrange][colrange].write("temppic");

        // Apply a point operator to output picture
        colrange += 128;
        pic[rowrange][colrange].point(thresh1);
        // Apply a point operator to other picture and show in output
        colrange += 128;
        temppic = other[otherrowrange][othercolrange];
        pic[rowrange][colrange].point(temppic,thresh2);
        // Apply a neigh operator to other picture and show in output
        colrange += 128;
        pic[rowrange][colrange].neigh(temppic,average);
        pic.show();
        pic.inspect();
        }
```

Using only simple assignment (e.g. `pic2[outrowrange][outcolrange] = pic1[inrowrange][incolrange];`) it is possible to move blocks of pictures around, stack pictures together, or cut out subpictures. For example, here is a program to extract a subimage of a `picture_of_int` and save it as a new picture. This program can easily be modified to work on `colour_pictures` too.

```
// Extract a subimage from a monochrome picture
#include "picture.h"
int main(int argc, char *argv[])
      {
      if (argc != 7) {
            cerr << "subimage inpic outpic xstart ystart xsize
ysize\n";
            exit(1);
            }
      int colstart = atoi(argv[3]);
      int rowstart = atoi(argv[4]);
      int ncols = atoi(argv[5]);
      int nrows = atoi(argv[6]);
      picture_of_int in(argv[1]);
      picture_of_int out(nrows, ncols);
      irange rowrange(rowstart,rowstart+1,rowstart+nrows-1);
      irange colrange(colstart,colstart+1,colstart+ncols-1);
      out = in[rowrange][colrange];
      out.write(argv[2]);
```

```
	return 1;
	}
```

Also using simple assignment, but this time assigning from an integer constant (e.g. `pic[rowrange][colrange] = 255`), it is possible to do efficient drawing of vertical and horizontal lines and filled blocks. Here, for example, is a program to overlay a mesh on a colour picture.

```
#include "picture.h"
// Overlays a green mesh on a picture
int main(int argc, char *argv[]) {
	if (argc != 4) {
		cerr << "Usage: overlaymesh inpic outpic meshspacing\n";
		exit(1);
		}
	colour_picture in(argv[1]);
	int spacing = atoi(argv[3]);
	irange allrows(0,1,in.nrows()-1);
	irange allcols(0,1,in.ncols()-1);
	irange spacedrows(0,spacing,in.nrows()-1);
	irange spacedcols(0,spacing,in.ncols()-1);
	// Draw horizontal lines
	if (in.ismonochrome())
		in.mono[spacedrows][allcols] = 255;
	else {
		in.green[spacedrows][allcols] = 255;
		in.red[spacedrows][allcols] = 0;
		in.blue[spacedrows][allcols] = 0;
		}
	// Draw vertical lines
	if (in.ismonochrome())
		in.mono[allrows][spacedcols] = 255;
	else {
		in.green[allrows][spacedcols] = 255;
		in.red[allrows][spacedcols] = 0;
		in.blue[allrows][spacedcols] = 0;
		}
	in.write(argv[2]);
	}
```

(By the way, if you forget to test whether the `colour_picture` is monochrome, CLIP is fairly forgiving. In the monochrome case, the green, red and blue buffers in the picture are all set up to point to the monochrome buffer, so whichever colour channel you write last is what the picture looks like.)

3.2 Block Matching with CLIP: A simple template matching example.

The way to add/subtract/etc. pictures and parts of pictures is by using the assignment operators +=, -=, etc.. We have already seen many examples like

```
	outpic += inpic;
```

and

```
	outpic[orowrange][ocolrange] -= inpic[irowrange][icolrange];
```

If you tried instead something like

```
	outpic = outpic + inpic;
```

you would not get the same kind of result. CLIP does support operators +, -, *, /, |, ^ between pictures, but the result of the operation is not a picture but rather an integer corresponding to the sum of all the pointwise additions/subtractions/etc.. In other words the result of

```
outpic + inpic
```

is a single integer representing the sum of all the pel-by-pel sums between `outpic` and `inpic` (which, of course, also equals the sum of all the pels in `inpic` plus the sum of all the pels in `outpic`). The statement

```
outpic = outpic + inpic;
```

sets every pel of `outpic` to this value. (Remember that assigning an integer to a `picture_of_int` or a `colour_picture` sets every pel to that integer value.)

Why do the operators +,-,*,/,|,^ work this counter-intuitive way? There are two reasons. The first is that to return a picture from an operation like + requires the construction of a temporary object, with all the allocation and deallocation of memory that that entails. So that would be inefficient. The second, more important, reason, is that CLIP's interpretation of +,-, etc. provides a powerful idiom for doing convolution, comparison, block matching, and other important operations.

CLIP provides both Sum Absolute Error and Sum Squared Error operators for matching blocks of a picture. Without further ado, here is a program that looks for the best match between a template picture and any number of other pictures. It does this by finding the position in the test picture where the Sum Absolute Error between the template and the picture is at a minimum. Enter this program, compile and debug it. We will then apply it to an experiment in template matching.

```
#include "picture.h"
int main(int argc, char *argv[]) {
      if (argc < 3) {
            cerr << "Usage: matchtemplate templatepic inpic1 inpic2 ...
\n";
            exit(1);
            }
      picture_of_int tpic(argv[1]); // Template picture
      picture_of_int inpic(argv[2]);
      inpic.show(argv[2]);
      irange trows(0,1,tpic.nrows()-1);
      irange tcols(0,1,tpic.ncols()-1);
      irange onepoint(0);
      for (int argnum = 2; argnum < argc; argnum++) {
            inpic.read(argv[argnum]);     // First time just re-reads
            int matchval;
            int bestmatchval =  100000000;      // A large number
            int besti, bestj;
            for (int i = 0; i < inpic.nrows(); i += 2) {
                  for (int j = 0; j < inpic.ncols(); j += 2) {
                        matchval = inpic[i+trows][j+tcols] | tpic;
                        // Above line gives absolute difference.
                        // Use ^ to get squared difference
                        if (matchval < bestmatchval) {
                              bestmatchval = matchval;
                              besti = i;
                              bestj = j;
                              }
                        }
```

```
            }
        // Draw a box at found location
        inpic[besti+trows][bestj+onepoint] = 0;
        inpic[besti+trows][bestj+tpic.ncols()+onepoint] = 0;
        inpic[besti+onepoint][bestj+tcols] = 0;
        inpic[besti+tpic.nrows()+onepoint][bestj+tcols] = 0;
        inpic.reshow(argv[argnum]);
        }
inpic.inspect();
}
```

Go to the ~/clip/sequence directory and display the picture stennis_000. Use di to determine a rough bounding box for the table tennis ball. Then use subimage to extract that region from stennis_000 as a small, monochrome template. Now try matchtemplate on the whole stennis sequence (Do this with the command "matchtemplate templatepic ../sequence/stennis*".) Observe how well the template fits the ball in different parts of the sequence. Consider (a) What could be done to improve accuracy? (b) How could frame-to-frame information be used (i.e. how could you *track* the ball rather than repeatedly detecting it)? (c) Why is this program so slow?! How could you speed it up?

3.3 Downsampling, upsampling, downsizing and upsizing.

Downsampling is the retention of every nth sample of a signal, throwing the rest away. Upsampling is one step in recovering an original signal from a downsampled version, by interposing the required number of zeros between samples. The programs downsample.cpp and upsample.cpp in ~/clip/bin/src do downsampling and upsampling by factors of two, and they can work horizontally only, vertically only, or in both directions at once. They rely on iranges, as you can see by reviewing the source code. To see how they work, you could run:

```
downsample inpic smallpic b e
upsample smallpic outpic b e
di inpic smallpic outpic
```

downsample appears to provide a quick way of halving the size of an image. But is it the best way? Should there be some pre-filtering of the input first, before samples are thrown away? Perhaps a quarter-sized image should be generated by taking the average of each block of four pels in the original image. What do you think?

upsample yields a sparse image of the same size as the original. What is the best way to fill in the gaps?

The program psnr is provided (in ~/clip/bin) to measure the Peak Signal to Noise Ratio between images. Usually it is used to compare a corrupted version of an image with the original. If you downsample an image, then upsample it as in the sequence above, then do "psnr inpic outpic" you will obtain a very low signal to noise ratio, because all the gaps in outpic are effectively noise not signal. You can measure the effectiveness of any gap filling by seeing how it improves the PSNR.

Downsampling is attractive for data compression. But it is only worth using if a close approximation to the original can be recovered. What we need is a sequence of operations like:

```
prefilter inpic filteredpic
downsample filteredpic smallpic b e
```
(Now we have achieved 4:1 data reduction by keeping smallpic only)
```
upsample smallpic sparsepic b e
postfilter sparsepic outpic
```

So the problem is to find a good prefilter and a good postfilter.

3.4 Geometric operations on pictures: Indexing with `doubles` and the `map()` function

downsample reduces the size of an image by removing every other sample. But what about scaling images to arbitrary sizes? You could just throw away samples irregularly (or repeat samples to increase image size), but a nicer solution is to interpolate values. CLIP will do this for you. It provides a way to index pictures using `doubles`, and automatically works out the bilinearly-interpolated values corresponding to the between-pel position that you specify. Here is a general-purpose scaling program that uses `double` indexing.

```
// Resize a picture by a given scale factor
#include "usage.h"
#include "picture.h"
#include "stdlib.h"
int main(int argc, char *argv[]){
      usage("resize inpic outpic scale");
      colour_picture in(argv[1]);
      double scale = atof(argv[3]);
      int outr = (int)(in.nrows()*scale);
      int outc = (int)(in.ncols()*scale);
      colour_picture out(outr, outc, 64, in.ismonochrome());
      if (in.ismonochrome()) {
            for (int i = 0; i < outr; i++)
            for (int j = 0; j < outc; j++) {
                  out.mono[i][j] = in.mono[i/scale][j/scale];
                  // The values i/scale and j/scale are doubles:
                  // CLIP automatically bilinear-interpolates
                  // from surrounding pels.
                  }
            }
      else {
            for (int i = 0; i < outr; i++)
            for (int j = 0; j < outc; j++) {
                  out.red[i][j] = in.red[i/scale][j/scale];
                  out.green[i][j] = in.green[i/scale][j/scale];
                  out.blue[i][j] = in.blue[i/scale][j/scale];
                  }
            }
      out.write(argv[2]);
      }
```

[Aside: This program uses the macro "`usage()`" which is defined in usage.h. `usage()` is not part of CLIP, but a simple macro I use to test that the program is called with the correct number of arguments, and print a usage message and exit if not. See usage.h for how it works. If you want to use `usage()`, be aware that the string argument that it takes must have the same number of whitespace-separated words as the correct usage command line.]

The above program, resize.cpp, is rather slow, because, for every output pel, there is not only a bilinear interpolation going on, but also the processing of four `operator[]` calls. CLIP therefore also provides a function `map()` that can be used instead of indexing with doubles. Using `map()` also means you don't need the if – else handling of monochrome and colour images. Here is the equivalent program, resize2.cpp, using `map()`.

```
// Resize a picture by a given scale factor
#include "usage.h"
#include "picture.h"
int main(int argc, char *argv[]){
      usage("resize inpic outpic scale");
      colour_picture in(argv[1]);
      double scale = atof(argv[3]);
      int outr = (int)(in.nrows()*scale);
      int outc = (int)(in.ncols()*scale);
      colour_picture out(outr, outc, 64, in.ismonochrome());
      for (int i = 0; i < outr; i++)
      for (int j = 0; j < outc; j++) {
            out.map(j, i, in, j/scale, i/scale, 0);
            }
      out.write(argv[2]);
      }
```

The arguments of `map()` are explained in picture.h, but of particular interest is the last argument which defines transparency. Whereas the &= operator is a blunt instrument for conditional replacement of pels in blocks, map provides 256 levels of transparency that can be used to combine pels from the two pictures concerned. In resize2.cpp the transparency argument is set to 0, meaning that nothing of the value that was in the picture out is retained. But with transparency = 255, almost all of the original value is retained. With transparency = 128, the old and new values are averaged.

Programs like resize can be used to generate geometric effects. Here is a program that uses repeated scaling to give the impression of a picture growing until it fills the window.

```
// Grow picture until it is full size
#include "usage.h"
#include "picture.h"
#include "stdlib.h"
int main(int argc, char *argv[]){
      usage("tunnel inpic");
      colour_picture in(argv[1]);
      colour_picture out(in.nrows(), in.ncols(), 64,
in.ismonochrome());
      double scale = 0.1;
      out.show(argv[1]);
      while (scale < 1) {
            int nr = (int)(in.nrows()*scale);
            int nc = (int)(in.ncols()*scale);
            int startr = (in.nrows() - nr)/2;
            int startc = (in.ncols() - nc)/2;
            for (int i = 0; i < nr; i++)
            for (int j = 0; j < nc; j++) {
                  out.map(j+startc, i+startr, in, j/scale, i/scale, 0);
                  }
            scale += 0.1;
            out.reshow();
            }
      out = in;
      out.reshow();
      out.inspect();
      }
```

CLIP does not directly support other geometrical operators – it's an Image Processing library rather than a Graphics library. However, it is reasonably easy to implement most important geometric transformations. Overleaf is a program, rotate.cpp, to rotate an image through a given angle. This illustrates coordinate transformation by multiplying with a 2 x 2 rotation matrix:

$$\begin{bmatrix} X \\ Y \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

```
// Program to rotate a picture about its centre
#include "picture.h"
#include "usage.h"
#include "math.h"
#include "stdlib.h"
int main(int argc, char *argv[]) {
      usage("rotate inpic outpic angle[degs]");
      colour_picture temp(argv[1]);
      int nr = temp.nrows();
      int nc = temp.ncols();
      int border = nr;
      if (nc > nr) border = nc;           // Very large border
      colour_picture in(argv[1],border);
      in.dc_pad();
      colour_picture out(in);
      double theta = atof(argv[3]);
      theta *= 3.141592/180;              // Convert to radians
      double sint = sin(theta);
      double cost = cos(theta);
      for (int i = 0; i < nr; i++)
      for (int j = 0; j < nc; j++) {
            int xout = j - nc/2;
            int yout = i - nr/2;
            double xin = xout*cost + yout*sint + nc/2;
            double yin = -xout*sint + yout*cost + nr/2;
            out.map(j, i, in, xin, yin, 0);
            }
      out.write(argv[2]);
      }
```

The new CLIP material in this program concerns the image border. Because the picture is going to be turned, parts of the border will become visible. Therefore it is important that the border is large enough. Note that when the `colour_picture` in is instantiated, a large border value is passed in explicitly as the constructor's second argument. Also, I choose to fill the border out so that, after rotation, the boundaries are still smooth. The line

```
      in.dc_pad();
```

accomplishes this.

3.5 Exercises

Do one exercise from part A and one from part B. Document both in your logbook. If you have time, try one exercise from part C.

Part A: Exercises based on questions in the tutorial.

1. Consider the experiment in Section 3.2. Suggest an answer to each of the questions posed there and investigate one of them by appropriate programming. The questions are: (a) What could be done to improve accuracy? (b) How could frame-to-frame information be used (i.e. how could you *track* the ball rather than repeatedly detecting it)? (c) Why is this program so slow?! How could you speed it up?

2. Consider the process discussed in Section 3.3. Conduct experiments to find a good prefilter/postfilter pair for downsampling and upsampling. You should be able to achieve a better PSNR than with

```
resize inpic smallpic 0.5
resize smallpic outpic 2
psnr inpic outpic
```

Part B: Extensions to programs in the tutorial.

3. Convert subimage.cpp into an interactive image cropping program.

4. Implement a program that moves sprites around a window using the &= operator.

5. Convert rotate.cpp into a program that spins an image through a full turn in small increments.

Part C: New programs

6. Many lenses introduce radial distortion into pictures. Develop a program that removes radial distortion by applying the following transformation to the image:

$$X = (1 - \alpha r^2)x$$
$$Y = (1 - \alpha r^2)y$$

where x, y, X and Y are relative to the centre of the picture, α is a user-specified parameter, and

$$r^2 = x^2 + y^2$$

Make the program interactive so a user can judge barrel/pincushion effects by eye.

7. Write a program that will draw and fill a given arbitrary quadrilateral. Aim for speed.

Adapt the program to warp a rectangular image onto the quadrilateral.

8. Consider the Paintshop Pro flow effects. These involve localized geometric warping. How would you implement a tool to rotate a small disk in the picture and have a surrounding annulus interpolate between the rotated disk and the unrotated remainder of the picture? Implement this.