CHAPTER 17

Image Processing

Introduction

In this chapter we're going to download and install an open-source library and then put it to work. The package I chose for this demonstration was Cyrus Harmon's "ch-image": a native Lisp library for representing, processing and manipulating images in a variety of formats. Unlike the libraries of Part III, this one is neither ubiquitous nor high on people's lists for standardization. What motivates this chapter-long example is:

- the steps described here are in some ways typical of the process for obtaining a third-party library;
- it's an informal opportunity to demonstrate numerical work in Lisp; and
- playing with pictures should come as light relief after the somewhat theoretical content of Chapter 16.

Ch-image is by no means the only graphics library for Lisp. It has the benefit of being small enough that we can summarise it in under 4000 words. People do cutting-edge image processing in Lisp—see for example the FREEDIUS geospatial and image compression toolkit from *http://www.ai.sri.com/software/freedius*—and many alternative libraries are available, up to and including a number of OpenGL bindings (for example), which we don't have space for here. This chapter covers the websites you should visit for locating Lisp libraries.

The first part of the chapter will show you how to set about capturing a library and integrating it into your code: very much part of the work of a professional programmer. The material which follows will be useful in setting a context for the rest of Part Four but it is not essential for understanding later chapters. The examples here employ a few standard Unix utilities such as `tar`, `ln -s` and `curl`; if you're not familiar with these then you could regard these examples as recipes; you certainly shouldn't let them put you off. I assume you know what an "RGB" image is. There's a bit of trigonometry in the longer example at the end of the chapter.

This chapter is peppered with suggestions to "look at the source", along with some exercises which involve using the inspector or debugger (whether explicitly, or implicitly when the code you've written goes wrong). If you're at all familiar with Emacs you might want to skip ahead and peruse the next chapter, on *SLIME* (the *Superior Lisp Interaction Mode for Emacs*), ahead of the end of this one. If you're doing any form of Lisp development work outside of one of the windowing GUIs then SLIME will make your life a lot easier.

Installation

I'm going to walk in detail through the tasks which I performed in order to install ch-image on a FreeBSD machine:

- locate the library I want to use;
- determine its requirements (dependent libraries, platform restrictions);
- download the appropriate bundles and unpack them as Lisp source files;
- compile and load the source into a Lisp session.

Downloadable software is a volatile beast and the specific issues I encounter here may not apply to ch-image even a year from now. Download instructions might have changed; platform dependencies might no longer require the use of an "unofficial" release; for all I know the whole library might have been totally refactored. On a more general level, what follows won't apply to all Lisp libraries and in some cases it's altogether more straightforward than this; it all depends very much on how the library was packaged up in the first place. We'll come back to downloads and installations in Chapter 19.

Having said all that, the general principles described here will remain valid even if the details do not, and as they will apply to many other Lisp libraries too the lesson won't be wasted.

If you try to take the exact steps that I did but find that links printed below have rotted, you'll find copies of everything I downloaded here:

http://lisp-book.org/3rd-party-libraries/

I've posted libraries there in two forms: the original download bundles, and as browsable files in case you'd rather peek at source code without downloading anything.

Locate

There is no single central repository for Lisp libraries.

There are however two good directory sites each listing hundreds of packages for a wide range of applications. When you're looking for a library you should normally consult both of these.

- The Common Lisp Directory at *http://www.cl-user.net/*
- The Common Lisp wiki (CLiki) at *http://www.cliki.net/*

I found ch-image on the CLiki by selecting the front page link for "Graphics Library interfaces" and then scanning a list of about 40 graphics-related entries until I found one which met my requirements. Two clicks from there brought me through to the project's home page:

http://cyrusharmon.org/projects?project=ch-image

Sometimes a direct search will be all you need. For example, when I needed to add compression to an application I'd been working on I simply googled for "lisp zip library". The top result in the search was David Lichteblau's "Common Lisp ZIP library" and I never needed to look any further.

I've also found that it's worth keeping half an eye on the metablog "Planet Lisp"

http://planet.lisp.org/

and making notes when people announce releases of libraries which look like they might turn out to be useful to me one day.

Examine

There is no centralized validation of Lisp libraries, no official stamp of approval.

Once you've found a candidate library, the next step is to determine whether it meets your requirements. There is no vetting procedure for work posted to any of the sites listed above and it's up to you to verify that the code you're proposing to use is fit for purpose and works the way you need it to, both

- in some general sense and,
- specifically, in co-operation with the rest of your software.

In the end the only way you're going to be able to do that is to try it out. In more complex cases—not this one—even basic testing might involve significant porting effort and so any preliminary investigations you can conduct become all the more valuable. (We'll give an example of such a port in Chapter 29.)

Start by checking the library's dependencies. According to the ch-image website, we need to download a total of six libraries:

- "ch-image" itself, from *http://cyrusharmon.org/static/releases/ch-image_0.4.1.tar.gz*

- "ch-util": a utility package with no documented dependencies, from *http://cyrusharmon.org/static/releases/ch-util_0.3.10.tar.gz*
- the "clem" matrix maths library, requires ch-util (above), available from *http://cyrusharmon.org/static/releases/clem_0.4.1.tar.gz*
- "cl-jpeg" which provides support for JPEG images: *http://www.common-lisp.net/project/mcclim/cl-jpeg.tar.gz*
- the "salza2" compression library, from *http://www.xach.com/lisp/salza2.tgz*
- and finally "zpng" which is for creating PNG files and itself uses salza2: *http://www.xach.com/lisp/zpng.tgz*

The first three are located by following links from the ch-image site. The others can be found instantly by googling on the library names.

Read the next section before downloading anything.

Platform

The next check is whether your platform is supported. While writing this section of the book I was working with version 1.3 of Clozure Common Lisp (CCL) on FreeBSD. The ch-image documentation does not mention hardware or operating systems and so we might assume that the code's only platform dependencies are on Lisp implementation. We could write to the author and ask for clarification; in any case we're going to test it all very shortly.

The documentation does state that the library was developed and tested on a different Lisp (SBCL) but that it "should run" on other implementations. A note on the main ch-image website labeled "clem and ch-image on non-SBCL lisps"

http://cyrusharmon.org/blog/display?id=95

supplies details and in particular download links which support CCL. So if you're working through this chapter with SBCL then the links listed above will work, and if like me you've got CCL in front of you they'll need some modification.

What follows isn't rocket science. The two links

http://git.cyrusharmon.org/cgi-bin/gitweb.cgi?p=ch-image.git

and

http://git.cyrusharmon.org/cgi-bin/gitweb.cgi?p=clem.git

take us to revision histories hosted under the *git* version control system (*http://git-scm.com/*). At the end of each entry is a link labeled "snapshot" whose URL is a tarball in .tar.gz format. The only question is whether to try the most recent revisions or the older but possibly more stable releases (clem 0.4.5 and ch-image 0.4.2). Both of these

are timestamped with a somewhat more recent date than the note which pointed us at them. Let's go for the releases; instead of the links printed above for ch-image and clem we'll use the release snapshots from the git repositories. The URLs are long and unilluminating so I've shortened them:

http://tinyurl.com/ch-image-0-4-2-tgz

and

http://tinyurl.com/clem-0-4-5-tgz

Download

We need to download and unpack our six libraries. A very effective Unix tool for the download is `curl` which in this simple form uses the `-O` option (upper case letter O) to read content associated with a URL and create a file with "the same name":

```
[ndl@vanity ~/chapter-17/tarballs]$ curl -O http://www.xach.com/lisp/salza2.tgz
.
.  (some output)
[ndl@vanity ~/chapter-17/tarballs]$ ls salza2.tgz
salza2.tgz
[ndl@vanity ~/chapter-17/tarballs]$
```

The TinyURLs above result in files with names as revolting as the snapshot URLs in the git repositories. Work round this with the `-o` option to `curl` to specify a name for the downloaded file:

```
curl -o ch-image_0.4.2.tar.gz http://tinyurl.com/ch-image-0-4-2-tgz
```

Once everything is downloaded, we have six tar files to decompress and unpack.

```
[ndl@vanity ~/chapter-17/tarballs]$ ls -l
total 1438
-rw-r--r--  1 ndl  ndl  1275408 Jul 28 13:52 ch-image_0.4.2.tar.gz
-rw-r--r--  1 ndl  ndl    10607 Jan 12  2008 ch-util_0.3.10.tar.gz
-rw-r--r--  1 ndl  ndl    21157 May 13  2008 cl-jpeg.tar.gz
-rw-r--r--  1 ndl  ndl    85104 Jul 28 13:58 clem_0.4.5.tar.gz
-rw-r--r--  1 ndl  ndl    15197 Jun 12 19:11 salza2.tgz
-rw-r--r--  1 ndl  ndl    39521 Sep 17  2008 zpng.tgz
[ndl@vanity ~/chapter-17/tarballs]$ cd ..
[ndl@vanity ~/chapter-17]$ tar -xfz tarballs/ch-image_0.4.2.tar.gz
[ndl@vanity ~/chapter-17]$ tar -xfz tarballs/ch-util_0.3.10.tar.gz
   (etc)
[ndl@vanity ~/chapter-17]$ ls -l
total 14
drwxr-xr-x  6 ndl  ndl   512 Nov  6  2008 ch-image
drwxr-xr-x  3 ndl  ndl  1024 Jul 28 14:08 ch-util_0.3.10
drwxr-xr-x  3 ndl  ndl   512 May 13  2008 cl-jpeg
drwxr-xr-x  6 ndl  ndl   512 Dec  3  2008 clem
drwxr-xr-x  3 ndl  ndl   512 Jun 12 19:08 salza2-2.0.7
drwxr-xr-x  2 ndl  ndl   512 Jul 28 13:59 tarballs
```

```
drwxr-xr-x  3 ndl  ndl   512 Sep 17  2008 zpng-1.2
[ndl@vanity ~/chapter-17]$
```

There's one last check to make before moving on. If you couldn't locate and hence examine licensing terms before you downloaded the libraries, do so now. (In this case all six libraries use permissive two- or three-clause BSD licenses.)

Congratulations! The nasty part is behind us and we can get on with the Lisp.

Build

The final task is to compile the source we downloaded and load the result into a running Lisp.

Working with system definitions—Lisp's answer to makefiles—and in particular with *ASDF* (*Another System Definition Facility*) is a major subject all on its own and we devote the whole of Chapter 19 to it. So for now let's just show how to build ch-image and load it into Clozure CL; we'll come back for explanations later.

Note that each of the six library directories contains a file named after that library and with extension *.asd*, for example *ch-image/ch-image.asd*. These are *system definition* files and the easiest way to proceed is to link to all of them from a single place:

```
[ndl@vanity ~/chapter-17]$ mkdir asdf; cd asdf
[ndl@vanity ~/chapter-17/asdf]$ ln -s ../*/*.asd .
[ndl@vanity ~/chapter-17/asdf]$ cd ..
```

(You'll spot that we ended up with about twice as many links as we were expecting here. Some of the download directories contain system definitions for documentation and test suites. These won't get in our way and so needn't concern us.)

This chapter is unashamedly Unix-flavored. Chapter 19 will discuss the use of ASDF on other platforms.

Now we fire up Lisp, tell it we need to use ASDF and where the .asd links are, and set it compiling:

```
[ndl@vanity ~/chapter-17]$ ccl
Welcome to Clozure Common Lisp Version 1.3-r12394M  (FreebsdX8632)!
? (require "asdf")
"asdf"
("ASDF" "asdf")
? (push "~/chapter-17/asdf/" asdf:*central-registry*)
("~/chapter-17/asdf/" *DEFAULT-PATHNAME-DEFAULTS*)
? (asdf:operate 'asdf:load-op "ch-image")
; loading system definition from /home/ndl/chapter-17/asdf/ch-image.asd ...
```

A few seconds and a couple of hundred lines of output later, we're back at the Lisp prompt. Most of this output consists of compiler warnings. We'll come back to these

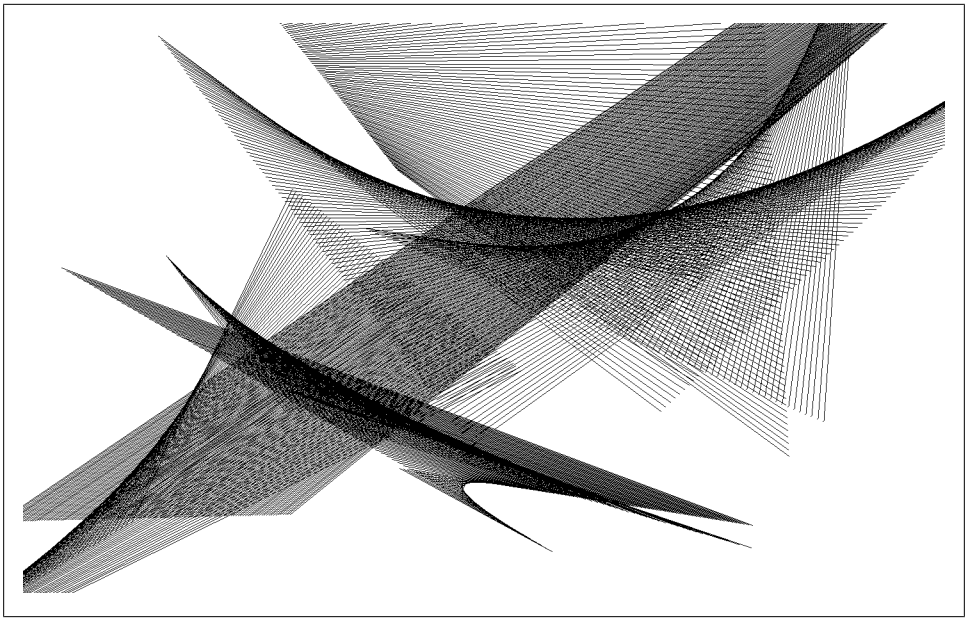


Figure 17-1. Random sets of lines: you might try this one for yourself. You'll need to make calls of the form (ch-image:draw-line image from-j from-i to-j to-i).

in Chapter 28 when we look at World Building. The important thing for now is to note that there aren't any errors in the report.

If you have to restart your Lisp session just repeat the three steps above (`require`, `push`, `asdf:operate`). This time it'll be even faster as there's no compiling to do—all Lisp has to do is load the compiled binaries—and there'll be considerably less output.

Exercise

Repeat the idiot mistake I made while researching this project and overlook the advice about how to work with CCL. Download release 0.4.1 of clem (i.e. the wrong version for Clozure CL) from its main website page (or from *http://lisp-book.org/3rd-party-libraries/*) and try to load it into CCL. Fix the major problem which arises. There's a hint at the end of the chapter. Don't spend too long on this if you're not making progress.

What's on Offer

This section is the software equivalent of kicking the tires and taking it out for a spin. There is some documentation on the project's website which is fine for getting started but incomplete. If we're not sure what a function is planning to do with its arguments

then a quick peek in the source might help. And there's absolutely no harm in trying things out.

Drawing Circles

```
(defun simple-circle (outfile)
  (let (;; Images are represented by CLOS instances. Make an instance of a
        ;; class which supports 8-bit RGB images.

        (image (make-instance 'ch-image:rgb-888-image
                              :height 200 :width 200)))

    ;; FILL-IMAGE sets the background color. The second argument represents
    ;; an RGB value. Turning red and green fully on and blue off gives us
    ;; yellow.

    (ch-image:fill-image image '(255 255 0))

    ;; The first two arguments to DRAW-CIRCLE specify its center, then
    ;; comes its radius and finally a foreground color.

    ;; WARNING: the ch-image package always specifies co-ordinates from the
    ;; top-left corner of the image and gives the row before the column.

    (ch-image:draw-circle image 100 100 80 '(0 0 255))

    ;; Write the image to our chosen destination. Note that we've built the
    ;; image without reference to an output format (.png in the invocation
    ;; below); the image's internal representation is totally generic.

    (values image
            (ch-image:write-image-file outfile image))))
```

```
? (simple-circle "round.png")
#<CH-IMAGE:RGB-888-IMAGE #x355C03DE>
#P"/home/ndl/chapter-17/round.png"
?
```

Exercise

Run this; find some way of viewing the result; adapt the example to make it more fun, or change it altogether (Figure 17-1).

The Common Lisp function `random` might help. `(random 8)` returns a pseudo-random integer between 0 and 7 inclusive; `(random pi)` returns a float between 0.0 and `pi`.

Exercise

Locate the source for `draw-circle` in *ch-image/src/shapes.lisp*. Identify the two tricks used to make this run faster than 360 degrees worth of naive Pythagoras invocations.

Image representation

The external image formats supported by ch-image are:

- JPEG,
- PNG (output only), and
- TIFF (SBCL only).

Two I/O functions are provided: `ch-image:write-image-file` as in the example above; and its input counterpart `ch-image:read-image-file`:

```
? (setf kings (ch-image:read-image-file "~/play/KingsCollegeChapel.jpg"))
#<CH-IMAGE:ARGB-8888-IMAGE #x34EA6496>
?
```

Both functions use the file's `pathname-type` to determine the appropriate external format.

Internally there are several different image formats to choose between including 8 and 16 bits-per-channel RGB and *ARGB* (that's RGB plus an "alpha" channel which specifies opacity) and 8, 16 and 32 bits-per-channel greyscale. The internal format of an image corresponds to its class, in particular:

- `ch-image:rgb-888-image` for 8-bit RGB (3 channels),
- `ch-image:argb-888-image` for 8-bit ARGB (4 channels), and
- `ch-image:ub8-matrix-image` for 8-bit greyscale (1 channel).

An image's channels are objects which you can manipulate directly, for example (see also Figure 17-2):

```
(defun read-call-write (function infile)
  (let* ((in-image (ch-image:read-image-file infile))
         (out-image (funcall function in-image))
         (outfile (make-pathname :name (format nil "~a-blue"
                                               (pathname-name infile))
                                 :defaults infile)))
    (values out-image
            (ch-image:write-image-file outfile out-image))))

(defun extract-blue-channel (original)
  (let* ((width (ch-image:image-width original))
         (height (ch-image:image-height original))
         (blue-only (make-instance 'ch-image:ub8-matrix-image
                                   :width width :height height)))
```

Figure 17-2. Kings College Chapel, Cambridge. An RGB image split by ch-image into its constituent channels. Clockwise from top left: red, green, blue. Lower left panel is greyscale generated from the RGB using the utility ch-image:argb-image-to-gray-image.

```
;; GET-CHANNELS and SET-CHANNELS work with a list of an image's
;; channels. IMAGE-R, IMAGE-G, IMAGE-B are individual accessors for the
;; three color channels.

(ch-image:set-channels blue-only (list (ch-image:image-b original)))

blue-only))
```

Exercise

Write an extension to this function which extracts all three color channels as separate outputs.

Exercise

Use either the SLIME inspector or `cl:inspect` to poke around inside a color image.

Pixels

The pixel values of each channel are numbers; for 8-bit images they're integers in the range 0-255 inclusive. You can also access pixel values without reference to channels. For a multi-channel image you'll get a list of numbers, for greyscale you get the number itself. Read and set pixel values thus:

```
? (ch-image:get-pixel kings 0 0)
(255 64 118 206)                          ; (alpha red green blue)
? (ch-image:set-pixel kings 0 0 '(255 46 118 206))
206
? (ch-image:set-pixel (ch-image:image-g kings) 0 0 181)
181
? (ch-image:get-pixel kings 0 0)
(255 46 181 206)
? (ch-image:get-pixel kings-blue 0 0)         ; a greyscale image
206
?
```

Exercise

The first call to `set-pixel` supplied four numbers and just one of them came back as the return value. What might you have expected as a more "obvious" value? Take a look at the source to figure out how this has come about. Write a simple method which fixes this (for all classes of image). Better still, implement `(setf ch-image:get-pixel)`.

Utilities

We've already met the function for drawing circles. Ch-image supplies utilities for:

- drawing circles, lines, triangles, rectangles, and general polygons;
- filling circles and rectangles;
- (SBCL only) generating text;
- masking one image according to pixel values in another;
- flipping an image horizontally;
- copying, cropping and resizing images.

Exercise

Visit ch-image's API documentation by following the link from the project webpage. Pick a drawing utility from first half of the above list and flick through the documentation to locate the function which supports it. If you find the results insufficiently informative, locate the source for this function and check what it does with its arguments. Get a call to this utility to work. Now repeat the experience with one of the higher-level tasks from the second half of the list.

At a higher level still, ch-image supports gamma correction, Gaussian blur, and image sharpening.

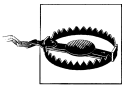
A Longer Example

Let's conclude our brief tour of ch-image with the full source of a small application for taking a perfectly respectable image and mangling it beyond recognition by applying a number of spiral deformations to it.

We define five functions. The devil is in the details and we comment on these inline. Note the inevitably repetitive nature of image processing code: it's hard to avoid saying everything twice (or, in `interpolate-pixel-values`, four times).

Exercise

Look ahead to the listing for `interpolate-pixel-values` at the end of this section. Using macros or otherwise make it less repetitive. Is the result any briefer? clearer? better?



Remember that, somewhat unusually, the ch-image package always specifies the row before the column. For consistency with the library we're using, we choose to follow that convention here and pass "j" values (verticals) before "i" values (horizontals).

```
;; Top-level utility. Applies a (small, random) number of spiral
;; deformations to the source image, each with randomly chosen
;; "strength" and centered about a random location. Sets each pixel in
;; the destination image by determining a source location for that
;; pixel and copying values from there. Returns deformed image.

(defun spiralize (source)
  (let* ((destination (ch-image:copy-image source))
         (width (ch-image:image-width source))
         (height (ch-image:image-height source))
         (max-i (- width 2))
         (max-j (- height 2)))
    ;; Loop with "random" parameters.
    (loop repeat (+ 3 (random 3)) do
          (let ((center-i (+ (round width 4) (random (round width 2))))
                (center-j (+ (round height 4) (random (round height 2))))
                (strength (* (+ 50.0 (random 100.0))
                             (if (zerop (random 2)) +1 -1)))
                ;; Make temporary copy of image.
                (temp (ch-image:copy-image destination)))
            ;; Crawl across destination image...
            (dotimes (destination-j height)
              (dotimes (destination-i width)
                ;; ... choosing a source pixel to copy into each
                ;; destination pixel ...
                (multiple-value-bind (source-j source-i)
```

Figure 17-3. The "Old Schools" building, Cambridge, distorted by spiralize.

```
                    (source-coordinates destination-j destination-i
                                        center-j center-i strength)
                  ;; ... and copy that pixel from the temporary source.
                  (copy-pixel temp
                              (max 0 (min source-j max-j))
                              (max 0 (min source-i max-i))
                              destination destination-j destination-i))))))
    destination))

;; Returns result of rotating co-ordinates (i, j) about given center
;; through angle proportional to strength and inversely proportional
;; to the distance between (i, j) and the center. New co-ordinates
;; need not be integers.

(defun source-coordinates (j i center-j center-i strength)
  (let* ((diff-i (- i center-i))
         (diff-j (- j center-j))
         (distance-squared (+ (* diff-i diff-i)
                              (* diff-j diff-j))))
    (if (zerop distance-squared)
        ;; Avoid dividing by zero.
        (values j i)
      (let* ((distance (sqrt distance-squared))
             (angle-to-rotate-by (/ strength distance)))
```

```
        (rotate j i angle-to-rotate-by center-j center-i)))))

;; Utility which rotates the point (i, j) about (center-i, center-j) by
;; theta radians.

(defun rotate (j i theta center-j center-i)
  (let* ((sin (sin theta))
         (cos (cos theta))
         (old-diff-i (- i center-i))
         (old-diff-j (- j center-j))
         (new-diff-i (- (* old-diff-i cos)
                        (* old-diff-j sin)))
         (new-diff-j (+ (* old-diff-j cos)
                        (* old-diff-i sin))))
    (values (+ center-j new-diff-j)
            (+ center-i new-diff-i))))

;; Utility for copying ch-image pixels. It iterates over channels and is
;; written to work equally with multi-channel images and greyscale.

(defun copy-pixel (source source-j source-i
                          destination destination-j destination-i)
  (loop for source-channel in (ch-image:get-channels source)
        as destination-channel in (ch-image:get-channels destination)
        do
        (let ((new-value (interpolate-pixel-values source-channel
                                                   source-j source-i)))
          (ch-image:set-pixel destination-channel
                              destination-j destination-i
                              new-value))))

;; Workhorse for averaging pixel values. We use bilinear interpolation,
;; exactly as in the clem package (see macro bilinear-interpolate in
;; "clem/src/interpolation.lisp"), to average values from four surrounding
;; source pixels.

(defun interpolate-pixel-values (source source-j source-i)
  (multiple-value-bind (base-i fraction-i)
      (floor source-i)
    (multiple-value-bind (base-j fraction-j)
        (floor source-j)
      (let ((pixel-00 (ch-image:get-pixel source  base-j      base-i))
            (pixel-01 (ch-image:get-pixel source  base-j     (1+ base-i)))
            (pixel-10 (ch-image:get-pixel source (1+ base-j)  base-i))
            (pixel-11 (ch-image:get-pixel source (1+ base-j) (1+ base-i))))
        (round ;; Apply "bilinear interpolation": the average of pixel
               ;; values is weighted according to how close we are to to
               ;; the location of each pixel.
               (+ pixel-00
                  (* fraction-i (- pixel-01 pixel-00))
                  (* fraction-j (- pixel-10 pixel-00))
                  (* fraction-i fraction-j (- (+ pixel-00 pixel-11)
                                              (+ pixel-01 pixel-10)))))))))
```

Exercise

Why do we make a fresh copy of the image `(let (... (temp (ch-image:copy-image destination))) ...)` each time round the loop in `spiralize`?

Exercise

The values `(- width 2)` and `(- height 2)` for `max-i` and `max-j` in `spiralize` don't look "right". What would you expect them to be, what goes wrong if you change them, and which function becomes more complicated if you want to "get it right"?

Exercise

What caused the stripy effect round the edges of Figure 17-3.

Exercise

Estimate how much memory is occupied by the internal representation of an image. Then see if this can be tallied to what `(room)` says.

Exercise

Rewrite this example to perform some totally different mangling operation on the source image. You'll have to modify `spiralize` and rewrite `source-coordinates`; you may or may not still want to use `rotate`; you probably won't have to touch `copy-pixel` or `interpolate-pixel-values`. Is it worth making `spiralize` "generic" (in the sense that the operation to be performed is passed to it as an argument)?

Hint for earlier exercise: the problem is in a package definition; once you've fixed that you need to make one other minor change to compensate; in total two one-line changes.