Tutorial for the cl-cairo2 package

Tamás K Papp

May 28, 2009

1 Introduction

Cairo is a 2D graphics library with support for multiple output devices. The `cl-cairo2` package provides Common Lisp bindings for the Cairo API.[1]

`cl-cairo2` is written with the following principles in mind:

CFFI bindings are generated by SWIG. This ensures that API changes are caught easily and makes it easier to follow them. The bindings to C functions are not exported directly.

It attempts to make the interface more Lisp-like. The Cairo API is written in C, which has no garbage collection or condition handling, and has little support for sophisticated dynamic data structures. However, the Lisp user should not worry about reference counting and pointer arithmetic. Instead of merely providing the C wrappers, cl-cairo2 aims to provide an interface conforming to the style of Lisp better.

Condition handling. In the Cairo API, an error is signaled by changing the state of the object, which the user is supposed to query periodically. The functions in cl-cairo2 do this automatically, and use Common Lisp's powerful condition facility to signal errors.[2]

This tutorial introduces cl-cairo2, but is not an introduction to Cairo itself. If you are not familiar with Cairo, it is recommended that your read the Cairo Tutorial for Python (and other) Programmers.

[1] Alternatives are cl-cairo, written by Lars Nostdal and others (this project appears to be dormant), and Christian Haselbach's cffi-cairo.

[2] This feature is not fully developed yet: currently a warning is signalled for all errors. The framework is in place, I just need to decide which errors require user intervention, etc.

2 Installation and loading

cl-cairo2 uses ASDF. Please refer to the ASDF and ASDF-Install manuals on how to install packages. You need to have the latest Cairo API installed. On Debian systems, you just need to install the `libcairo2` package. You don't need the header files or SWIG unless you plan to regenate the SWIG bindings.

Once installed, you can load cl-cairo2 with

```
(asdf:operate 'asdf:load-op :cl-cairo2)
```

3 Getting started

Most Cairo drawing operations are performed on a *context*. Think of the context as a combination of a canvas which remembers pen strokes (the current path), color, line width, line style, and other, more complicated settings that determine what gets drawn where. When you build a path (which you want to fill with a color/pattern, stroke with a given line style, or even save), all of this happens in a context.

All `cl-cairo2` functions take the context as their (optional) last argument. It defaults to `*context*`. The macro `with-context` will wrap your code in a `(let ((*context* ...) )...)` with the given context. When you are writing your own functions that implement more complex operations, I recommend that you make `context` as an optional or keyword argument, with `*context*` as its default value.

Contexts are created from *surfaces* — which, at this point, should be thought of as the bare canvas itself (think of PDF, PostScript, of PNG files). All Cairo objects, including contexts and surfaces, are implemented in CLOS wrappers, and can be closed (*destroyed*) with `destroy`.

When the context is created from a surface, the reference count (in the internals of Cairo) of the latter is incremented. You can immediately destroy the surface: it will not be destroyed (ie the file will not be closed) until you destroy the context.[3] The following code draws a white diagonal line on a blue background, using a Postscript file – the result is shown in Figure 1.

```
(defparameter *surface* (create-pdf-surface "example.pdf" 200 100))
(setf *context* (create-context *surface*))
(destroy *surface*)
;; clear the whole canvas with blue
```

[3]The file will also be closed if the wrapper object is garbage collected. However, you should not rely on this, as calling the garbage collector is not portable.

```
(set-source-rgb 0.2 0.2 1)
(paint)
;; draw a white diagonal line
(move-to 200 0)
(line-to 0 100)
(set-source-rgb 1 1 1)
(set-line-width 5)
(stroke)
;; destroy context, this also destroys the surface and closes the file
(destroy *context*)
```

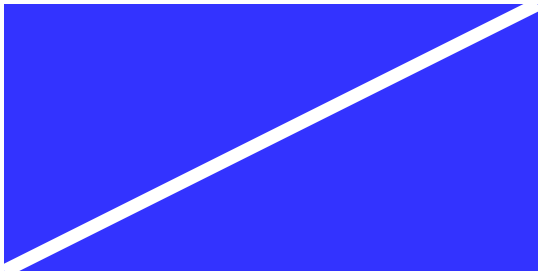

Figure 1: white diagonal line on a blue background

Unless you need the surface itself for something else, you should use the `create-*-context` convenience commands provided by cl-cairo2. For example, the first three lines of the code above would be replaced by

```
(defparameter *context* (create-ps-context "example.ps" 200 100))
```

Unlike the original Cairo API, surfaces and contexts in `cl-cairo2` remember their width and height. Use the generic functions `get-width` and `get-height` to extract these.

When you want to write the output into a bitmap file (for example, in PNG format), you first need to create an *image surface*, then write this to the bitmap file when you are done. The macro `with-png-file` will take care of these details: use it like

```
(with-png-file ("example.png" :rgb24 200 100)
  ;; drawing commands
  ...)
```

This example above highlights another feature of cl-cairo2: constants (such as the format specifier `rgb24`) are exported as simple names from the cl-cairo2 package. Internal functions in the package map these to the enum constants used by Cairo. cl-cairo2 uses

lookup tables (assoc lists) for this purpose, which are defined in `tables.lisp`. Cairo constants `CAIRO_PROPERTY_SOMETHING` usually map to the Lisp keyword symbol `:property-something`, and can only be used in setting or querying `PROPERTY`. For example, `CAIRO_FORMAT_RGB24` is mapped to `:format-rgb24`, and using it for some other property will create an error.

Likewise, names of the Lisp function are easy to deduce from the name of the C function in the Cairo API: just drop the `cairo_` prefix and convert all underscores (`_`) to dashes (`-`). The exceptions to this rule (and the explanations) are given in Table 3.

Cairo API name (explanation)	cl-cairo2 name
`cairo_fill` (would conflict with `cl:fill`)	`fill-path`
`cairo_identity_matrix` (would conflict with matrix algebra packages)	`reset-trans-matrix`
`cairo_matrix_init_identity`	use `(make-trans-matrix)`
`cairo_matrix_transform_distance`	`transform-distance`
`cairo_matrix_transform_point`	`transform-point`
`cairo_matrix_*`	`trans-matrix-*`

4 Implementation notes

4.1 General

The package contains some helper functions, most notably `deg-to-rad`, which converts degrees to radians. Cairo functions use the latter.

4.2 Surfaces

See the beginning of `surface.lisp` for helper macros used internally to define wrappers for the SWIG-generated CFFI interface (neither the interface not these macros are exported). `with-alive-surface` checks if the pointer for a surface object is nil, and `check-surface-pointer-status` queries the status of the surface after executing `body`. `with-surface` is a combination of the two, and `new-surface-with-check` makes a new surface object from a pointer, checking its status first.

Currently, only Postscript, PDF, SVG and image surfaces (which can be written to PNG files) are supported.

Drawing in X11 windows is implemented using the `x11-context` class — see Section 4.7 for more information. You need to load the `cl-cairo2-x11` package for that.

4.3 Contexts

Contexts are represented as the class `context`, which currently only has one slot, a pointer to the context. When contexts are destroyed, this is set to `nil`.

The macro `with-context-pointer` is similar to `with-surface` above (it executes the body with pointer pointing to the object and then checks error status). The functions `define-with-default-context` defines a function acting on a context given a list of arguments and exports this function. `define-flexible` is similar, but allows a more flexible function body.

Functions that are not implemented yet include

- `cairo-get-target`
- `push-group-with-content`
- `get-group-target`
- `set-source`, `set-source-surface`, `get-source`
- `mask`, `mask-surface`

I doubt that Lisp users need `get/set-user-data` or `get-reference-count`. Let me know if you do.

Since version 0.2.3, you can use colors from cl-colors with the generic function `set-source-color`, for example,

```
(set-source-color +darkolivegreen+)
```

4.4 Paths

Almost all drawing operations of Cairo rely on the construction of paths. While basic paths can be constucted in a context, paths can be saved, modified and reused independently. At the moment, cl-cairo2 does not support these path operations:

- `copy-path`
- `copy-path-flat`
- `path-destroy`
- `append-path`

This is because I decided to follow the recommendation of relevant section of the Cairo API Manual, and implement paths and related manipulators in a way that doesn't require the user to traverse structures of C pointers. This is not done yet.

`glyph-path` (see Section 4.5 for discussion about glyphs) is not implemented either.

4.5 Text

Text operations are very basic at the moment (refer to `text.lisp` for an enumeration of what is missing). You can select font face and size using commands like

```
(select-font-face "Arial" :italic :bold)
(select-font-size 12)
```

and use `(show-text "hello␣world")` to draw it. You can control text placement by using `text-extents` and recalculating the position — see `example.lisp` for examples (in Section 5). Note that Cairo functions accept text in UTF-8 format: you should convert your strings to UTF-8 if you plan to use non-ASCII characters.

The long-term goal is to use CLOS for font selection, following the recommendations here. Also, proper handling of glyphs would be a nice thing, but would require other libraries (eg Pango) for converting text to glyphs.

4.6 Transformations

cl-cairo2 defines the structure `trans-matrix` with the slots `xx`, `yx`, `xy`, `yy`, `x0`, `y0`. The defaults for these slots give you the identity matrix.

All the functions that use transformation matrices use this structure. Consequently, `cairo_matrix_init` has no corresponding function in cl-cairo2: you can construct a translation matrix using `make-trans-matrix`.

Some functions are renamed, see Table 3. Generally, functions which manipulate `trans-matrix` stuctures start with `trans-matrix-`, and other a few other functions have been renamed to avoid conflicts with linear algebra packages.

4.7 Xlib Contexts

The xlib context is not part of cairo – it is a bit of glue code that uses cairo's X11 surface on a pixmap, and displays this pixmap when needed (when X11 asks for the window contents to be redrawn or when cairo draws on the pixmap). The X11 specific code is in a separate package, so make sure that you load `cl-cairo2-x11`.

Please remember that the X11 code provided is a proof of concept, only for displaying the results of Cairo commands interactively in windows. If you would like to use Cairo in your — possibly more complex — applications, you need to do things differently (for example, you need you own event loop).

Two contexts are implemented, one uses double-buffered pixmaps, and can be created by `create-xlib-context`, the other Xlib image surfaces, and you can create such a context by `create-xlib-image-context`. I suggest that you use the latter.

In cl-cairo2, each window maps to a context. The surface is not exposed to the user, who is only allowed to see the context. This makes memory management and proper cleanup easier. For example, you can create an `xlib-image-context` with

```
(setf *context* (create-xlib-image-context 500 400
                                           :display-name "localhost:0"
                                           :window-name "my␣pretty␣drawing"))
```

If you give `nil` for `display-name`, Xlib fill probably figure out a reasonable default, usually from your `$DISPLAY` environment variable. You can also specify the background color.

The X11 event loop runs in a separate thread, so you need a Lisp implementation that supports threads.

When the context `destroy`ed, the window is closed. This works the other way too: when the window is closed, the context is destroyed. The implementation precludes the resizing of the window.

The current implementation is not optimized for speed (the whole window is redrawn all the time) but it is fast enough. If you draw a lot of objects at the same time, it is suggested that you suspend synchronizing with the X-window server using `(sync-lock context)`. When you are done, you can call `(sync-unlock context)`, which will automatically sync the buffer and the window. You can nest calls to `sync-lock` and `sync-unlock`, and if you want to restore syncing unconditionally, use `sync-reset`, which also performs syncing too. These are generic functions which do nothing for other contexts.

4.8 MS-Windows contexts

Kei Suzuki contributed code for interfacing Cairo to MS-Windows. Please see `package-win.lisp`, you need to load the `cl-cairo2-win` library to use this code.

4.9 To Do

The list below reflects my priorities. If you need something, please let me know.

- CLOS integration for fonts (as suggested here)

Things I don't plan on doing, but will be happy to incorporate if somebody does it:

- Pango and/or glyph handling

5 Examples

Below is an extended example, which can be found in `example.lisp`. Figures 2–4 show the results.

```
(require :asdf)
(asdf:operate 'asdf:load-op :cl-cairo2)

;;;; Make a test package
(defpackage :cairo-example
  (:use :common-lisp :cl-cairo2))

(in-package :cairo-example)

;;;;
;;;; pathname where files will end up, you need to change it to
;;;; something that makes sense on your system if you are running
;;;; these examples
;;;;

(setf *default-pathname-defaults* #P"/home/tpapp/software/cl-cairo2/
   tutorial/")
(assert (directory *default-pathname-defaults*))

;;;;
;;;; short example for the tutorial
;;;;

(defparameter *surface* (create-pdf-surface "example.pdf" 200 100))
(setf *context* (create-context *surface*))
(destroy *surface*)
;; clear the whole canvas with blue
(set-source-rgb 0.2 0.2 1)
(paint)
;; draw a white diagonal line
(move-to 200 0)
(line-to 0 100)
(set-source-rgb 1 1 1)
(set-line-width 5)
(stroke)
;; destroy context, this also destroys the surface and closes the file
(destroy *context*)
```

```
;;;; very simple text example
(setf *context* (create-pdf-context "simpletext.pdf" 100 100))
(move-to 0 100)
(set-font-size 50)
(show-text "foo")
(destroy *context*)


;;;;
;;;; text placement example
;;;;
;;;; This example demonstrates the use of text-extents, by placing
;;;; text aligned relative to a red marker.

;;;;
;;;; helper functions
;;;;

(defun show-text-aligned (text x y x-align y-align &optional
                          (context *context*))
  "Show␣text␣aligned␣relative␣to␣(x,y)."
  (with-context (context)
    (multiple-value-bind (x-bearing y-bearing width height)
        (text-extents text)
      (move-to (- x (* width x-align) x-bearing)
               (- y (* height y-align) y-bearing))
      (show-text text))))

(defun mark-at (x y d red green blue &optional (context *context*))
  "Make␣a␣rectangle␣of␣size␣2d␣around␣x␣y␣with␣the␣given␣colors,
␣␣50%␣alpha.␣␣Used␣for␣marking␣points."
  (with-context (context)
    (rectangle (- x d) (- y d) (* 2 d) (* 2 d))
    (set-source-rgba red green blue 0.5)
    (fill-path)))

(defun show-text-with-marker (text x y x-align y-align &optional (context *
   context*))
```

```
  "Show␣text␣aligned␣relative␣to␣a␣red␣market␣at␣(x,y)."
  (with-context (context)
    (mark-at x y 2 1 0 0)
    (set-source-rgba 0 0 0 0.6)
    (show-text-aligned text x y x-align y-align)))

(defparameter width 500)
(defparameter height 500)
(defparameter text "Fog")  ; contains g, which goes below baseline
(defparameter size 50)
(defparameter x 20d0)
(defparameter y 50d0)
(setf *context* (create-pdf-context "text.pdf" width height))
;; white background
(set-source-rgb 1 1 1)
(paint)
;; setup font
(select-font-face "Arial" :normal :normal)
(set-font-size size)
;; starting point
(mark-at x y 2 1 0 0)  ; red
;; first text in a box
(multiple-value-bind (x-bearing y-bearing text-width text-height)
    (text-extents text)
  (let ((rect-x (+ x x-bearing))
        (rect-y (+ y y-bearing)))
    (rectangle rect-x rect-y text-width text-height)
    (set-source-rgba 0 0 1 0.3)  ; blue
    (set-line-width 1)
    (set-dash 0 '(5 5))
    (stroke)))
(set-source-rgba 0 0 0 0.6)
(move-to x y)
(show-text text)
(dolist (x-align '(0 0.5 1))
  (dolist (y-align '(0 0.5 1))
    (show-text-with-marker text (+ x (* x-align 300))
                           (+ y (* y-align 300) 100)
```

```
                        x-align y-align)))
;; done
(destroy *context*)



;;;;
;;;; Lissajous curves
;;;;

(defparameter size 500)
(defparameter margin 20)
(defparameter a 9)
(defparameter b 8)
(defparameter delta (/ pi 2))
(defparameter density 2000)
(setf *context* (create-pdf-context "lissajous.pdf" size size))
;; pastel blue background
(rectangle 0 0 width height)
(set-source-rgb 0.9 0.9 1)
(fill-path)
;; Lissajous curves, blue
(labels ((stretch (x) (+ (* (1+ x) (- (/ size 2) margin)) margin)))
  (move-to (stretch (sin delta)) (stretch 0))
  (dotimes (i density)
    (let* ((v (/ (* i pi 2) density))
           (x (sin (+ (* a v) delta)))
           (y (sin (* b v))))
      (line-to (stretch x) (stretch y)))))
(close-path)
(set-line-width .5)
(set-source-rgb 0 0 1)
(stroke)
;; "cl-cairo2" in Arial bold to the center
(select-font-face "Arial" :normal :bold)
(set-font-size 100)
(set-source-rgba 1 0.75 0 0.5)  ; orange
(show-text-aligned "cl-cairo2" (/ size 2) (/ size 2) 0.5 0.5)
```

```
;; done
(destroy *context*)

;;;;
;;;; transformation matrix example (for Judit, with love)
;;;;
;;;; This example uses the function heart which fills a heart-shape
;;;; with given transparency at the origin, using a fixed size.
;;;; Rotation, translation and scaling is achieved using the
;;;; appropriate cairo functions.

(defun heart (alpha &optional (context *context*))
  "Draw␣a␣heart␣with␣fixed␣size␣and␣the␣given␣transparency␣alpha.
␣␣Heart␣is␣upside␣down."
  (with-context (context)
    (let ((radius (sqrt 0.5)))
      (move-to 0 -2)
      (line-to 1 -1)
      (arc 0.5 -0.5 radius (deg-to-rad -45) (deg-to-rad 135))
      (arc -0.5 -0.5 radius (deg-to-rad 45) (deg-to-rad 215))
      (close-path)
      (set-source-rgba 1 0 0 alpha)
      (fill-path))))

(defparameter width 1024)
(defparameter height 768)
(defparameter max-angle 40d0)
(setf *context* (create-pdf-context "hearts.pdf" width height))
;; fill with white
(rectangle 0 0 width height)
(set-source-rgb 1 1 1)
(fill-path)
;; draw the hearts
(dotimes (i 200)
  (let ((scaling (+ 5d0 (random 40d0))))
    (reset-trans-matrix) ; reset matrix
    (translate (random width) (random height)) ; move the origin
    (scale scaling scaling)  ; scale
```

```
    (rotate (deg-to-rad (- (random (* 2 max-angle))
                                   max-angle 180))) ; rotate
    (heart (+ 0.1 (random 0.7)))))
(destroy *context*)

;;;;
;;;; make a rainbow-like pattern
;;;;
;;;;

(defparameter width 100)
(defparameter height 40)
(setf *context* (create-pdf-context "pattern.pdf" width height))
(with-linear-pattern rainbow (0 0 width 0)
    `((0 (0.7 0 0.7 0)) ;rgb(a) color as list
      (1/6 ,cl-colors:+blue+) ;color as cl-color
      (2/6 ,cl-colors:+green+)
      (3/6 ,cl-colors:+yellow+)
      (4/6 ,cl-colors:+orange+)
      (5/6 ,cl-colors:+red+)
      (1 ,cl-colors:+violetred+))
  (rectangle 0 0 width height)
  (set-source rainbow)
  (fill-path))
(destroy *context*)

;;;;
;;;; example for with-png-file
;;;;
(with-png-file ("simple.png" :rgb24 200 100)
  ;; clear the whole canvas with blue
  (set-source-rgb 0.2 0.2 1)
  (paint)
  ;; draw a white diagonal line
  (move-to 200 0)
  (line-to 0 100)
  (set-source-rgb 1 1 1)
  (set-line-width 5)
```

```
(stroke))
```

Figure 2: text.pdf

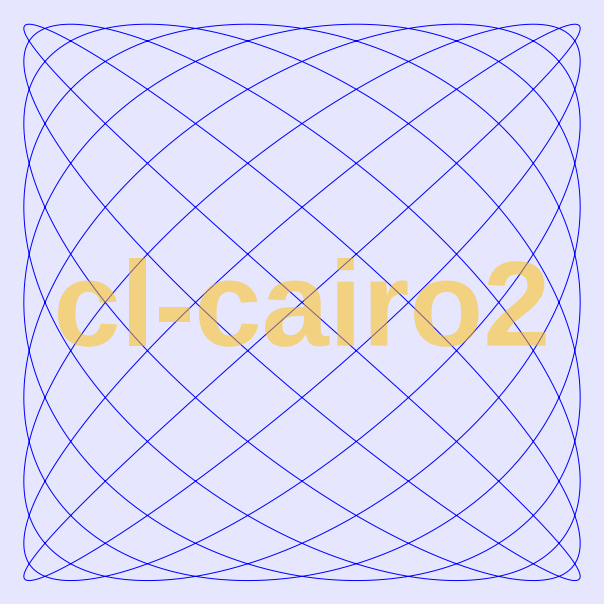

Figure 3: lissajous.pdf

Figure 4: hearts.pdf

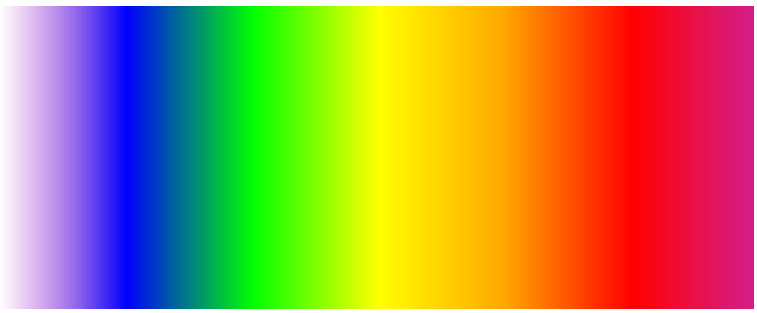

Figure 5: pattern.pdf