IBM English Sign in (or register

developerWorks Technical topics Evaluation software Community Events

Cross-platform graphics with cairo

Vector drawing library aims to produce consistent output

Eli M. Dow (emdow@us.ibm.com), Software Engineer, IBM

Summary: Built from the ground up to create identical output on both printer and screen—all in a cross-platform way—cairo is becoming a huge player in the Linux® graphics space. Harness the same 2D power used by GNOME, GTK+, Pango, and many others.

Date: 05 Sep 2007
Level: Intermediate
Also available in: Russian Japanese

Activity: 13147 views
Comments: 0 (View | Add comment - Sign in)

Average rating (13 votes)
Rate this article

Cairo benefits and uses

Cairo is a free software vector drawing library that can draw to multiple output formats. Cairo provides support for a rich set of platforms including Linux, BSDs, Microsoft® Windows®, and OSX (a BeOS and OS2 backend are also being developed). Linux drawing can be accomplished via the X Window system, Quartz, image buffer formats, or OpenGL contexts. Additionally cairo allows you to render graphics to PostScript or PDF output for high-quality print results. Ideally, users of cairo will get as near identical outputs from print and on screen as possible.

This article shows you what cairo is, and why it may be useful in your applications. The example shown here will produce a pdf, ps, png, svg, and gtk window displaying a version of the IBM logo.

A significant design decision in cairo is to support nearly identical output to the greatest extent possible. This consistent output lends itself exceptionally well for GUI toolkit programming, or cross-platform application development. The ability to print a screen at high resolution, and draw on the screen contents with the same drawing library has obvious advantages.

Further, on each supported target, cairo attempts to make intelligent use of the underlying hardware and software support. The combination of high-quality vector graphics and high performance makes cairo well suited to becoming the modern UNIX® drawing system of choice.

Cairo is written in C, but has bindings for most common languages. The C language choice aids in new binding creation, while simultaneously allowing high performance when invoked natively. Worthy of special mention are the Python bindings, which enable rapid prototyping, as well as a lower bar of entry for those learning the cairo drawing API.

Vector drawing vs. bitmap drawing

Cairo is a **vector drawing** library and thus the drawing revolves around algebraic descriptions of a drawing rather than the sequence of filled pixels of a bitmap. With **bitmap drawing**, a series of pixels are filled with predetermined color values in a predetermined arrangement, and the quality of the drawing is proportional to the size of the bitmap.

The bitmap method of drawing breaks down when you zoom into or magnify a bitmap image. Often images look "fuzzy" or blurred, giving an effect similar to that of walking up close to a rear projection or other large television. At a distance, the image may look clear, but close up you can see the sequence of individual points. A distinct loss of definition exists, as no data can define what should exist between the previously defined pixels.

Computer drawing systems and architectures have been around for quite some time, and cairo owes much of its design to the earlier PostScript and PDF models. Cairo has modeled the PostScript and Portable Document Format (PDF) approach because they both use mathematical statements to define the image. This algebraic notation of the image allows the entire image, or just a portion of the image, to be recreated at any time by evaluating the algebraic description over the range of interest. The algebraic nature is expressed as points, curves, and lines (these comprise the vectors that give vector drawing its name).

Because the mathematics can be re-evaluated to redraw the image or its components, no loss in resolution occurs when you zoom, scale, or transform the image. However, there are some limits to vector drawing. For instance, in some cases scaling a vector image to extremely high zoom values, beyond what most persons would ever want to do practically, can introduce glitches. Zooming in can cause some lines to appear incorrectly due to rounding errors in the calculations. When zooming out, certain lines may become invisible or imperceptible.

An added benefit to the vector nature of cairo drawing is that vector images tend to be smaller in size. This is because a relatively large amount of information can be encoded in a relatively small equation. The beauty of vector drawing is that the drawing tends to be relatively straightforward. The onus of actually converting the points, lines, and their associated equations into something you can see rests on the drawing library.

The equations describing these curves are known as **Bezier curves** or paths, named after the mathematician Pierre Bezier. The Bezier curve consists of at least 2 anchor points, as well as one or more additional points between them known as **handles**. A handle point may be moved to alter the shape of the curve. If you've used tools like Photoshop or the GIMP, you may have played with these types of curves. However, the final format of your saved drawing may have been a bitmap! The format of the file determines if the Bezier path information is preserved, or if it is to be evaluated over a given range and saved as a bitmap.

As of this writing, available cairo bindings include C++, Ruby, Perl, the Java™ language, and .Net/mono, among others. Each of the bindings is in a separate state of developmental maturity; please refer to the cairo project home page (see Resources for a link) for up-to-the-minute details on each. Right now, the Python and C++ bindings seem to be used most extensively in the open source community at large.

As mentioned previously, several graphics toolkits provide bindings to make cairo development even easier. Gtk+ versions newer than 2.8 contain full support for cairo, and cairo has been selected as the strategic drawing system to support future GTK releases. Additionally, toolkits like GNUstep and FLTK are beginning to support cairo for their graphics rendering needs.

Cairo makes perfect sense to select as your drawing API if you plan on doing anything cross-platform that requires low-level control of drawing operations and compositing. And if you want to have the cross-platform capabilities but do not want to draw at a low level, there are some other convenience drawing libraries that sit on top of cairo.

Why learn a new imaging model?

Quite frankly, I found existing open source solutions to be lacking in various ways. While xprint had the advantage of a unified display and print API, it was typically configured to run as a separate server process and had a poor API. And while libgnomeprint had separate print and imaging models, the separation of print and imaging APIs causes rendering variances between the screen and printer.

Cairo learned from its predecessors and has been designed from the ground up to implement a common API.

Cairo rendering targets

Cairo can render to the following output formats:

- X Window System (utilizing the Render extension when available)
- OpenGL (through the use of the glitz backend)
- In-Memory Images (pixbuffs and so on)
- PostScript (suitable for printing)
- PDF (Portable Document Format) files
- SVG (Scalable Vector Graphics) format

But not all rendering targets are created equal. Though cairo strives to create identical outputs on the varying backends, each backend has its own set of advantages. For example, the PDF backend is vector-based where possible (only falling back to images when necessary), whereas the PostScript backend essentially generates one large image per page.

The rendering model presented in cairo has been influenced by a number of pre-existing technologies. Cairo has notions of paths, strokes, and fills from PostScript, and has implemented Porter-Duff image compositing from PDF and from render extensions in modern X server implementations. Additionally cairo implements the normal complement of clipping, masks, and gradients.

Applications of cairo in the wild

A large number of influential open source projects have jumped on the cairo bandwagon, and cairo has positioned itself to be a huge player in the Linux graphics space. Some of the more influential projects that are already embracing cairo are:

- Gtk+, everyone's favorite cross platform graphics toolkit
- Pango, a Free Software library for laying out and rendering text, with emphasis on internationalization
- Gnome, a Free Software desktop environment
- Mozilla, cross-platform Web browser infrastructure on which Firefox is based
- OpenOffice.org, a free software office suite comparable to Microsoft Office

---

Conceptual drawing with cairo

When drawing with cairo, the simplest operations are akin to choosing your medium, selecting a brush, choosing a color, thinking about the line placement, then actually drawing the lines. The cairo folks use a painter analogy in their own documentation, so I will use the same here below.

Typically the easiest thing for a painter to do is select his or her blank medium. In the physical world, an artist may choose to draw on paper, cloth, or even a wall. In cairo you must also select your blank medium. When drawing in this way you need to set up a cairo context, which is the main object. From this context you can select your target surface such as a PostScript file, PDF document, or on-screen image. In this way you can select what we want to paint on.

Now let's consider the painter's next task of selecting a brush. A painter can spend considerable time selecting the appropriate brush tips and sizes. With cairo there is also a notion of a brush tip expressed through the stroke width. Different stroke widths produce lines of different thicknesses.

Now, unlike an artist painting in the real world, a user of cairo needs to speak in terms of exact coordinates. An artist might just place the brush to paper, but computers need x and y coordinates to know where to place the digital paint.

Once cairo knows about the paintbrush and location you wish to begin drawing from, you need to imagine what the stroke will look like. Simple drawings might be straight lines, but you are free (just as a real artist is) to draw curves and arcs.

And finally you must define the position where the stroke is to terminate. Once again, this is specified using an (x,y) coordinate pair.

You may additionally want to "color in" the objects you have drawn. In cairo terminology, this is called a **fill**. For each of the operations described above, there is a straightforward corresponding API implementation in cairo. Some of these introductory APIs are described below.

These basic operations can help you build some very complex graphics. Cairo even allows you to do things that painters cannot easily do, like take your existing drawing and apply transformations that equate to zooming in or moving your drawing somewhere else on the virtual paper.

While GIMP or Photoshop allow you to perform many of these same actions, remember that cairo is different: cairo is a programmatic means of writing art. GIMP and Photoshop use tools like cairo "under the hood" to implement their drawing. In drawing with those tools, the mouse automatically sets the coordinate points and the type of tool, such as a box, and selects the pen and stroke for you via a GUI environment. As you can see in the sample code (see Download), cairo requires more explicit interaction, on the order of "use a stroke width of 1 set an arc with radius 10 about a centerpoint at location z."

Proper cairo terminology

It is important to use correct terminology whenever discussing any technology. Cairo's API terminology fits in 3 categories: core drawing terms, surface terms, and font-related terms (see Resources for more details).

First and foremost, cairo has a drawing context, akin to the drawing motions performed on the painter's canvas from the earlier analogy. The context is of type cairo_t and is necessary for rendering your drawings. A common thing to do on such a context is to render Bezier shapes, lines and curves. The cairo terminology for such a series of curvilinear shapes and associated data is a **path**. These paths may be drawn, and subsequently stroked or filled.

Coordinates are transformed into paths using a very simple API. This API is excellent because it frees you from having to think about the

kinds of complex transformation matrices commonly discussed in linear algebra or graphics courses and textbooks. The drawing operations found in cairo can be transformed by any affine transformation. These transformations give you the tools to shear, scale, or rotate your drawings or parts of drawings. Each path is drawn by specifying points. In this way cairo operates in a connect-the-dots fashion. We will see an example of this later.

Next we will discuss the various cairo surface types. Several kinds of cairo surfaces exist, and each maps to the target output for the given drawing. Cairo **surfaces** are the thing that you draw to. Specifically, surfaces exist for images (in-memory buffers), Open GL via the glitz surface, PDF and PostScript surfaces for document rendering, and direct platform drawing via the XLib and Win32 surfaces. Each of these surfaces derives from a base surface type, cairo_surface_t.

In cairo, a **pattern** is something that you can read from, that is used as the source or mask of a drawing operation. Patterns in cairo can be solid, surface-based, or even gradients patterns.

So far, we have only talked about stroking paths. But path stroking generally produces line drawings that are uninteresting. In fact, simple line stroking is only one of the 5 essential drawing operations in cairo. The 5 operations are:

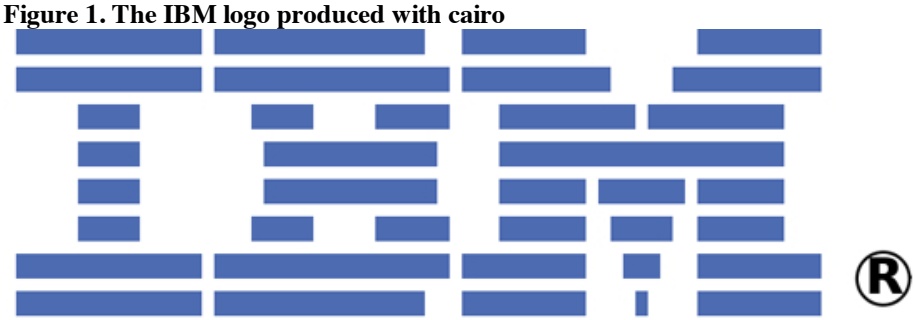
- cairo_stroke
- cairo_fill
- cairo_show_text/cairo_show_glyphs
- cairo_paint
- cairo_mask

Though simple line drawings can be convenient, they are simply not expressive enough for something as complicated as fonts. Cairo has a base class for fonts called cairo_font_face_t. Cairo understands scaled fonts, which contain caching metrics for a given font size. Additionally, various font options can dictate how a given font is to be rendered. Commonly, the fonts used when dealing with cairo are Freetype fonts on UNIX, and Win32 fonts on Windows platforms.

A sample cairo application

I wrote a bit of code that uses cairo to paint the IBM logo. You can download the code from the Download section below. Running it should produce the following output:

Figure 1. The IBM logo produced with cairo



In the code, note the line `cairo_stroke (cr)`, which appears after I've drawn the letters but before I've added the "Registered" mark. Cairo will not draw anything without a stroke, and accidentally failing to include the stroke is a common problem for beginners.

Cairo changes in the pipeline

Cairo release versions follow a pattern similar to that of the Linux kernel—that is, odd-numbered releases are experimental, development versions unsuitable for the faint of heart, or those with production environments to maintain. Even-numbered releases are more polished: the original 1.0 release focused on getting the API right for users, and making high-quality output. The 1.2 API focused on completing a few of the less-fully-developed backends, while the current 1.4 series has made much progress in optimization and new functionality.

The cairo developers have provided some nice sample code snippets showing off various pieces of the cairo API (see the link to those sample programs in the Resources section below). So don't wait for the next cairo release; go ahead and download the current version and try it out!

Download

Description	Name	Size	Download method
Cairo code to draw IBM logo	cairo-example.tar	20KB	HTTP

Information about download methods

Resources

Learn

- The cairo graphics project home page includes a cairo user manual and sample code snippets illustrating various parts of the cairo API, and much more.
- See an up-to-the-minute list of language and toolkit bindings for cairo including those for Objective Caml and GNUstep.
- Check out "Writing a Widget Using Cairo and GTK+2.8" a *GNOME Journal* article.
- An illustrated glossary of common vector graphics terms is provided by the Technical Advisory Service for Images (or TASI).
- In the developerWorks Linux zone, find more resources for Linux developers, and scan our most popular articles and tutorials.
- Stay current with developerWorks technical events and Webcasts.

Get products and technologies

- Order the SEK for Linux, a two-DVD set containing the latest IBM trial software for Linux from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- With IBM trial software, available for download directly from developerWorks, build your next development project on Linux.

Discuss

- Check out developerWorks blogs and get involved in the developerWorks community.

About the author

Eli M. Dow is a software engineer in the IBM Test and Integration Center for Linux in Poughkeepsie, NY. He holds a B.S. degree in computer science and psychology and a master's of computer science from Clarkson University. He is an alumnus of the Clarkson Open Source Institute. His interests include the GNOME desktop, human-computer interaction, virtualization, and Linux systems programming. He is the coauthor of an IBM Redbook *Linux for IBM System z9 and IBM zSeries*.

Close [x]

developerWorks: Sign in

IBM ID:
Need an IBM ID?
Forgot your IBM ID?

Password:
Forgot your password?
Change your password

Keep me signed in.

By clicking **Submit**, you agree to the developerWorks terms of use.

Submit Cancel

The first time you sign into developerWorks, a **profile** is created for you. **Select information in your developerWorks profile is displayed to the public, but you may edit the information at any time**. Your first name, last name (unless you choose to hide them), and display name will accompany the content that you post.

All information submitted is secure.

Close [x]

Choose your display name

The first time you sign in to developerWorks, a profile is created for you, so you need to choose a display name. Your display name accompanies the content you post on developerWorks.

Please choose a display name between 3-31 characters. Your display name must be unique in the developerWorks community and should not be your email address for privacy reasons.

Display name: (Must be between 3 – 31 characters.)

By clicking **Submit**, you agree to the developerWorks terms of use.

Submit Cancel

All information submitted is secure.

Average rating (13 votes)

1 star 1 star
2 stars 2 stars
3 stars 3 stars
4 stars 4 stars
5 stars 5 stars

Submit

Add comment:

Sign in or register to leave a comment.

Note: HTML elements are not supported within comments.

Notify me when a comment is added1000 characters left

Post

Be the first to add a comment

Print this page Share this page Follow developerWorks

About
Help
Contact us
Submit content

Feeds and apps
Newsletters

Report abuse
Terms of use
IBM privacy
IBM accessibility

Faculty
Students
Business Partners