# vbRichClient6 (RC6) & RC6 Widgets

# for Dummies (like me)

**[](https://www.cairographics.org/)**

Updated 9/6/2022 – Work in Progress

**Cairo Tutorial**

**Table of Contents**

* Forward – Why this Documentation?
* [Cairo's Drawing Model](https://www.cairographics.org/tutorial/#L1draw)
  + [Nouns](https://www.cairographics.org/tutorial/#L2nouns)
  + [Verbs](https://www.cairographics.org/tutorial/#L2verbs)
* [Drawing with Cairo](https://www.cairographics.org/tutorial/#L1drawing)
  + [Preparing and Selecting a Source](https://www.cairographics.org/tutorial/#L2preparesource)
  + [Creating a Path](https://www.cairographics.org/tutorial/#L2createpath)
* [Understanding Text](https://www.cairographics.org/tutorial/#L1understandingtext)
* [Working with Transforms](https://www.cairographics.org/tutorial/#L1transforms)
* [Where to Go Next](https://www.cairographics.org/tutorial/#L1gonext)
* [Tips and Tricks](https://www.cairographics.org/tutorial/#L1tips)
  + [Line Width](https://www.cairographics.org/tutorial/#L2linewidth)
  + [Text Alignment](https://www.cairographics.org/tutorial/#L2textalign)

IMPORTANT: Be sure to add a reference to RC6.dll in your project.

This unauthorized abomination of a tutorial document was written by WEBBIZ.

**Forward – Why this documentation?**

If the title is not enough to answer this question, then I will elaborate here. I am not a professional programmer, and I want to get that out right now. In fact, I know even less about vbRichclient6, the RC6 Widgets, and Cairo 2D graphics. So why would you want to read, let alone use, this document?

Perhaps this will help you decide. As I am writing this information I am currently incorporating this framework into a VB6 application I wrote sometime around the turn of this century (21st). As was the case when I originally wrote the application, each modification or feature is created ‘on-the-fly’. What that means is that when I have something that needs to be done, I learn just enough to do it (often with the help of others online), then move on. Well, that also happens to be the case in incorporating this framework into a very old VB6 application.

I have spent an incredible amount of hours trying to learn how to use this framework for my VB6 project. Reading code, the included comments when available, and doing multiple searches online or within the vbForum, is a bit of a choppy way to learn for a greenhorn programmer. You find yourself heavily reliant on the assistance of others, which means a lot of waiting time. So as I accumulate the answers, why not document it for future reference? And in the event another dummy like me comes around needing more hand-holding, wouldn’t it be nice to have something like this around?

So why bother? Why not use the MS API and its GDI/GDI+ libs? Because I like anything that makes my work simpler, and I am going to guess you do as well or you would not be reading this.

Olaf Schmidt is the person behind this framework. As he has described on his website (<http://www.vbrichclient.com>), “It’s main purpose is to decouple from as many MS-COM dependencies as possible…”. There is more to it, and I invite you to visit his website where you will also find the latest version of the RC framework.

In addition to providing a great wrapper for those interested in SQLite (years ago Olaf provided much help in my incorporating SQLite in another app that still has users today), he has binded the Cairo 2D graphics library originally written in the C programming language into RC6 making it easily accessible to VB6 programmers. I invite you to check out the Cairo site at (<https://cairographics.org>).

While I cannot possibly include everything that this framework can do, enough should be included here to give a leg up. You will likely need help anyway, but if it saves you time along the way, then it has been worth it. As for me, I may need to refresh my brain cells on this subject, and I know I stored some of those cells here. ☺

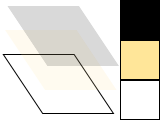
**Cairo's Drawing Model**

In order to explain the operations used by cairo, we first delve into a model of how cairo models drawing. There are only a few concepts involved, which are then applied over and over by the different methods. First I'll describe the [nouns](https://www.cairographics.org/tutorial/#L2nouns): [destination](https://www.cairographics.org/tutorial/#L3destination), [source](https://www.cairographics.org/tutorial/#L3source), [mask](https://www.cairographics.org/tutorial/#L3mask), [path](https://www.cairographics.org/tutorial/#L3path), and [context](https://www.cairographics.org/tutorial/#L3context). After that I'll describe the [verbs](https://www.cairographics.org/tutorial/#L2verbs) which offer ways to manipulate the nouns and draw the graphics you wish to create.

**Nouns**

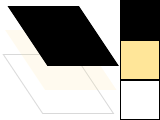
Cairo's nouns are somewhat abstract. To make them concrete I'm including diagrams that depict how they interact. The first three nouns are the three layers in the diagrams you see in this section. The fourth noun, the path, is drawn on the middle layer when it is relevant. The final noun, the context, isn't shown.

**Destination**



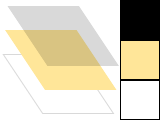
The destination is the [surface](http://www.cairographics.org/manual/cairo-surfaces.html) on which you're drawing. It may be tied to an array of pixels like in this tutorial, or it might be tied to a SVG or PDF file, or something else. This surface collects the elements of your graphic as you apply them, allowing you to build up a complex work as though painting on a canvas.

**Source**



The source is the "paint" you're about to work with. I show this as it is—plain black for several examples—but translucent to show lower layers. Unlike real paint, it doesn't have to be a single color; it can be a [pattern](http://www.cairographics.org/manual/cairo-cairo-pattern-t.html) or even a previously created destination [surface](http://www.cairographics.org/manual/cairo-surfaces.html) (see [How do I paint from one surface to another?](http://cairographics.org/FAQ/#paint_from_a_surface)). Also unlike real paint it can contain transparency information—the Alpha channel.

**Mask**



The mask is the most important piece: it controls where you apply the source to the destination. I will show it as a yellow layer with holes where it lets the source through. When you apply a drawing verb, it's like you stamp the source to the destination. Anywhere the mask allows, the source is copied. Anywhere the mask disallows, nothing happens.

**Path**

The path is somewhere between part of the mask and part of the context. I will show it as thin green lines on the mask layer. It is manipulated by path verbs, then used by drawing verbs.

**Context**

The context keeps track of everything that verbs affect. It tracks one source, one destination, and one mask. It also tracks several helper variables like your line width and style, your font face and size, and more. Most importantly it tracks the path, which is turned into a mask by drawing verbs.

Before you can start to draw something with cairo, you need to create the context. When you create a cairo context, it must be tied to a specific surface—for example, an image surface if you want to create a PNG file. You can initialize your cairo context like this:

‘First declare a surface variable

Dim Srf As cCairoSurface

‘Now create the surface object and assign to variable

Set Srf = Cairo.CreateSurface(120, 120, ImageSurface)

The cairo context in this example is tied to an image surface of dimension 120 x 120. Surfaces can be created specific to most cairo backends.

**Surfaces**

[**ImageSurface**](https://www.cairographics.org/manual/cairo-Image-Surfaces.html)**— Rendering to memory buffers**

[**PDFSurface**](https://www.cairographics.org/manual/cairo-PDF-Surfaces.html)**— Rendering PDF documents**

[**PSSurface**](https://www.cairographics.org/manual/cairo-PostScript-Surfaces.html)**— Rendering PostScript documents**

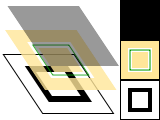
[**SVGSurface**](https://www.cairographics.org/manual/cairo-SVG-Surfaces.html)**— Rendering SVG documents**

**DXSurface32Bit -**

**Verbs**

The reason you are using cairo in a program is to draw. Cairo internally draws with one fundamental drawing operation: the source and mask are freely placed somewhere over the destination. Then the layers are all pressed together and the paint from the source is transferred to the destination wherever the mask allows it. To that extent the following five drawing verbs, or operations, are all similar. They differ by how they construct the mask.

**Stroke**

[](https://www.cairographics.org/tutorial/stroke.c)

The context .Stroke operation takes a virtual pen along the path. It allows the source to transfer through the mask in a thin (or thick) line around the path, according to the pen's [line width](http://www.cairographics.org/manual/cairo-cairo-t.html#cairo-set-line-width), [dash style](http://www.cairographics.org/manual/cairo-cairo-t.html#cairo-set-dash), and [line caps](http://www.cairographics.org/manual/cairo-cairo-t.html#cairo-set-line-cap).

‘First define surface and assign to variable

Dim Srf As cCairoSurface

Set Srf = Cairo.CreateSurface(300, 300, ImageSurface)

‘Now declare a variable for the context

Dim CC As cCairoContext

‘Associate context to surface

Set CC = Srf.CreateContext

‘Define a pixel path to represent your shape

‘Here I define a rectangle path starting at (20,20)

‘100 pixels wide, 75 pixels down

CC.Rectangle 20, 20, 100, 75

‘Set a color for the path

CC.SetSourceColor vbGreen

‘Set a path (line) width

CC.SetLineWidth 5

‘Now we can create the path onto the surface that

‘has been defined above

CC.Stroke

‘In order to display it on the form, pass the form hDC

‘to the surface object draw

Srf.DrawToDC hDC

\*\*This document will use <cCairoContextObject> to represent the context object variable (like CC above). For example: <cCairoContextObject>.Stroke.

**Fill**

[](https://www.cairographics.org/tutorial/fill.c)

The context .Fill operation instead uses the path like the lines of a coloring book, and allows the source through the mask within the hole whose boundaries are the path. For complex paths (paths with multiple closed sub-paths—like a donut—or paths that self-intersect) this is influenced by the [fill rule](http://www.cairographics.org/manual/cairo-cairo-t.html#cairo-set-fill-rule). Note that while stroking the path transfers the source for half of the line width on each side of the path, filling a path fills directly up to the edge of the path and no further.

‘Using the previous example, instead of drawing

‘a line with .stroke, you can fill the inside of

‘a closed path.

CC.Rectangle 20, 20, 100, 75

‘set your line width, color, etc. and then use

CC.Fill

**Show Text / Glyphs**

[](https://www.cairographics.org/tutorial/showtext.c)

The <cCairoContextObject>.TextOut(x As Double, y As Double, Text As String, [BaseLine As Boolean], [Alpha As Double = 1], [PathOnly As Boolean]) operation forms the mask from text. For word wrap and different Font-Rendering qualities (CAIRO\_ANTIALIAS\_NONE, CAIRO\_ANTIALIAS\_DEFAULT, CAIRO\_ANTIALIAS\_SUBPIXEL, CAIRO\_ANTIALIAS\_BEST, CAIRO\_ANTIALIAS\_GOOD, CAIRO\_ANTIALIAS\_GRAY), there is <cCairoContextObject>.DrawText(x As Double, y As Double, dx As Double, dy As Double, S As String, [SingleLine As Boolean], [Alignment As AlignmentConstants], [InnerSpace As Long], [VAlign As Long], [ExtendedFlags As DTExtFlagsEnm], [Alpha As Double = 1], [PathOnly As Boolean], [MaxDxNeeded As Double]) As Long

‘Select a font, font-size and color

CC. SelectFont “Times New Roman”, 25, vbRed

‘Create the text

CC.TextOut 25, 30, “Hello World!”

‘Display it

Srf.DrawToDC hDC, 20, 20

‘Select a font, font-size and color

CC.SelectFont “Times New Roman”, 25, vbRed

‘Optional render quality

Cairo.FontOptions = CAIRO\_ANTIALIAS\_DEFAULT

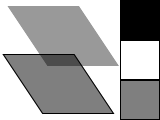
‘Draw the text

CC.DrawText 0, 0, 300, 300, “This is a text box and this text will wrap inside this box based on next parameter set to False.”, FALSE, vbLeftJustify

‘Display it

Srf.DrawToDC hDC, 20, 20

**Paint**

[](https://www.cairographics.org/tutorial/paint.c)

The <cCairoContextObject>.Paint([Alpha As Double = 1], [Pattern As cCairoPattern]) operation uses a mask that transfers the entire source to the destination. Some people consider this an infinitely large mask, and others consider it no mask; the result is the same. The Alpha parameter similarly allows transfer of the full source to destination, but it transfers only the provided percentage of the color, with the default 1 (100%) as opaque.

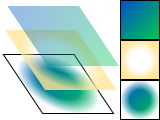
‘Define a surface color

CC.SetSourceColor vbGreen

‘Paint the surface with 60% opaque (ie. 40% transparency)

CC.Paint .6

**Mask**

[](https://www.cairographics.org/tutorial/mask.c)

The <cCairoContextObject>.Mask(Pattern As cCairoPattern) operations allow transfer according to the transparency/opacity of a second source pattern or surface. Where the pattern or surface is opaque, the current source is transferred to the destination. Where the pattern or surface is transparent, nothing is transferred. This will allow the ability to create a clipping or gradient mask.

The following code does not expose the full power of the Mask routines available, but is a simple demonstration. Another routine (not shown) is <cCairoContextObject>.MaskSurface(Pattern As cCairoPattern, [srfX as Double], [srfY as Double]).

‘You will need a surface and a Context associated with

‘it. Here you can do both at once.

Dim CC As cCairoContext

Set CC = Cairo.CreateSurface(800, 600).CreateContext

‘Set the color

CC.SetSourceColor vbRed

‘Now we need a pattern for the mask. Here is a quick

‘way to create a radial pattern and apply to mask.

With Cairo.CreateRadialPattern(0, 0, 0, 0, 0, 320) ‘radial stamp

.AddGaussianStops\_TwoColors 0, 0, 1, 0 ‘Alpha\_Channel only

CC.Mask .This ‘Uses radial pattern for stamping

End With

Picture = CC.Surface.Picture ‘Assign Surface as Form Picture.

**Drawing with Cairo**

In order to create an image you desire, you have to prepare the [context](http://www.cairographics.org/manual/cairo-cairo-t.html) for each of the drawing verbs. To use <cCairoContextObject>.Stroke or <cCairoContextObject>.Fill you first need a path. To use <cCairoContextObject>.Mask you need a second source [pattern](http://www.cairographics.org/manual/cairo-cairo-pattern-t.html) or [surface](http://www.cairographics.org/manual/cairo-cairo-surface-t.html). And to use any of the operations, including <cCairoContextObject>.Paint, you need a primary source.

**Preparing and Selecting a Source**

There are three main kinds of sources in cairo: colors, gradients, and images. Colors are the simplest; they use a uniform hue and opacity for the entire source. You can select these without any preparation with <cCairoContextObject>.SetSourceRGB and <cCairoContextObject>.RGBA. Using <cCairoContextObject>.SetSourceRGB (R As Double, G As Double, B As Double) is equivalent to using <cCairoContextObject>.RGBA (R As Double, G As Double, B As Double, [A As Double = 1]), and it sets your source color to use full opacity. The following code example demonstrates these methods in order to create the image you see here:

[](https://www.cairographics.org/tutorial/setsourcergba.c)

Option Explicit

Private Sub Form\_Load()

Me.ScaleMode = vbPixels

Me.AutoRedraw = True

'Create a surface

Dim Srf As cCairoSurface

Set Srf = Cairo.CreateSurface(Me.ScaleWidth, Me.ScaleHeight, ImageSurface)

'Create a context associated with the surface created above

Dim CC As cCairoContext

Set CC = Srf.CreateContext

CC.ScaleDrawings Srf.Width, Srf.Height

CC.Paint 1, Cairo.CreateSolidPatternLng(vbWhite)

CC.SetSourceRGB 0, 0, 0

CC.MoveTo 0, 0

CC.LineTo 1, 1

CC.MoveTo 1, 0

CC.LineTo 0, 1

CC.SetLineWidth 0.2

CC.Stroke

CC.Rectangle 0, 0, 0.5, 0.5

CC.SetSourceRGBA 1, 0, 0, 0.8 '80% opacity (alpha)

CC.Fill

CC.Rectangle 0, 0.5, 0.5, 0.5

CC.SetSourceRGBA 0, 1, 0, 0.6 '60% opacity (alpha)

CC.Fill

CC.Rectangle 0.5, 0, 0.5, 0.5

CC.SetSourceRGBA 0, 0, 1, 0.4 '40% opacity

CC.Fill

CC.TranslateDrawings 100, 100

Set Me.Picture = Srf.Picture

'Srf.WriteContentToPngFile "d:\setsourcergba.png"

'Srf.WriteContentToJpgFile "d:\jumping.jpg"

End Sub

I will attempt to explain what is happening in the code shown above.

When working with Cairo drawing, you want to make sure you have your ScaleMode set to vbPixels. The .AutoRedraw is set TRUE so that the form can be refreshed each time it has been covered by another window or changed in some way (ex: resized).

Before we can draw we need a **surface** to draw on. We can create this (destination) surface by declaring a cCairoSurface variable (Dim Srf As cCairoSurface) and then create an object from cCairoSurface and assign it to that variable (Set Srf = Cairo.CreateSurface(…) ).

There are methods we will want to have access to in order to draw lines, rectangles, set the line width, scale the dimension of the drawing area, etc. Therefore, we need to create a **context** that has these methods and associate it with the surface object.

Start by declaring a cCairoContext variable that will hold the surface context object.

Dim CC As cCairoContext

Now we must create that context object using the .CreateContext() method of the surface object itself so that the context will be associated with that surface.

Set CC = Srf.CreateContext

Great! Now we have a **surface** to draw on and the **context** that will provide us with the methods we need to draw.

Cairo uses a coordinate system that represents the drawing space dimensions from 0 to 1, much like saying ‘from 0 to 100%’ width or height. Thus, the top of the surface area is considered 0 and the bottom 1. The left axis is 0 and the right 1. So 0,0 would be top left and 1,1 would be bottom right.

However, we need to define our drawing surface width and height values to match the pixels we will be working with. In other words, if our drawing surface is to be 100 pixels wide and 200 pixels high, we will need to define the SCALE so that if 1 (100%) will represent 100 or 200, depending on whether it is the x or y coordinate.

In the example code above, we simply set the x,y coordinates to the surface width and height values using this line:

**CC.ScaleDrawings Srf.Width, Srf.Height**

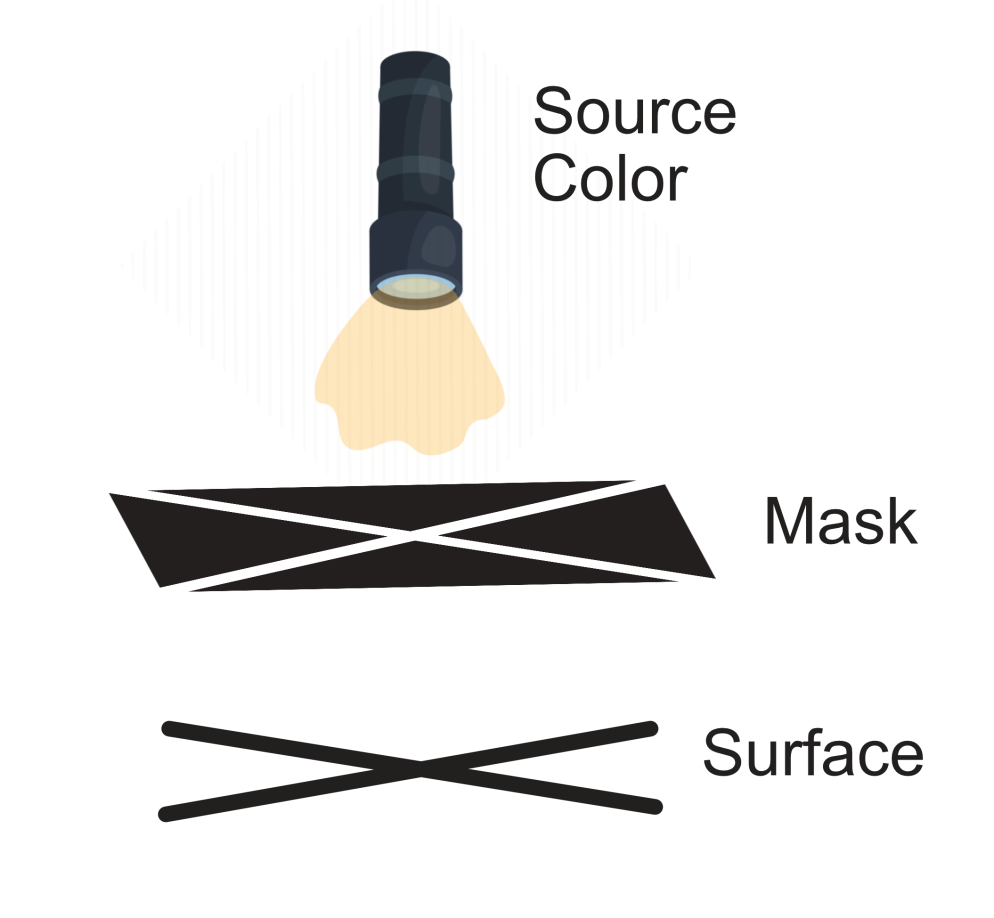
Next we will paint the surface area itself with a solid pattern of white. If we don’t do this, it defaults to black and our diagonal lines will not show up as they are also black.

**CC.Paint 1, Cairo.CreateSolidPatternLng(vbWhite)**

Now we need to set the source color for our lines we are going to draw. This can be done using the following code that sets the draw color to black.

**CC.SetSourceRGB 0, 0, 0**

As discussed in the beginning of the document, how drawing is done is based on a Source, Mask, and then Destination (Surface) layer. So think of it like this:

****

When you use instructions such as **CC.MoveTo 0, 0 and CC.LineTo 1, 1**, you are creating a PATH (opening) on the MASK layer (see above). This PATH is from 0, 0 (left, top) to 1, 1 (right, bottom). For our example, this draws a diagonal line from the top-left 0 to the bottom-right (Srf.Width, Srf.Height).

To complete the X pattern, we need another diagonal line. This time we will draw from right-top to left-bottom. We do this with **CC.MoveTo 1, 0 and CC.LineTo 0, 1**. This becomes right (Srf.Width), top (0) to left (0), bottom (Srf.Height).

Now the PATH for an X pattern is complete. However, we do need to define the width of our PATH lines. To do this, we use:

**CC.SetLineWidth 0.2**

The parameter for .SetLineWidth is also based on 0 to 1.

Up until now, we have set the SOURCE color and we have created a PATH onto the MASK. But figuratively, we have not turned on the source light yet. To do this in order to project our newly draw path onto the surface, we need to use either a STROKE (this only draws along the PATH itself, with half the width on one side and half the width on the other side of the line, much like straddling the line) or use a FILL (this fills everything within the lines and only up to the lines). In our example, the diagonal X is drawn with the STROKE.

**CC.Stroke**

This is like turning on the source, imprinting the black X onto the surface. But of course this is just an illustration to help visualize.

Now we move on to drawing some rectangles onto our surface. Same procedure, we start by defining the PATH that will be created onto the MASK. I will describe the first rectangle as the other two will be similar.

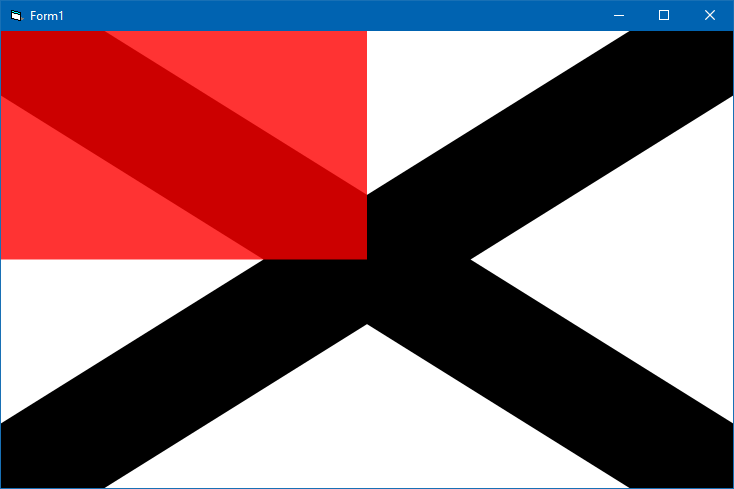
**CC.Rectangle 0, 0, 0.5, 0.5**

Note that here we see 0 as left, 0 as top, 0.5 as halfway from left to right, and 0.5 as halfway from top to bottom. What you get is a rectangle that covers the upper-left quarter of the surface. The color is defined as RED using:

**CC.SetSourceRGBA 1, 0, 0, 0.8**

Remember, 1 equates to 100%. Since this is RGB, the first parameter is R (red) and the number 1 means 256 (RGB values are 0 to 256). Green and Blue are 0 so you effectively have RGB(256, 0, 0) which is RED.

The 0.8 is the Alpha channel setting, which is set to 80% of full opacity. CC.Fill then allows the source through the mask to appear on the surface.



In order to visually see our surface result we can use the following line:

Set Me.Picture = Srf.Picture

That is all you need to do in order to draw basic shapes and view them. If you wish to save it in a PNG or JPG file for example, you can use the following:

Srf.WriteContentToPngFile "d:\setsourcergba.png"

This is for saving a PNG file called ‘setsourcergba.png’ onto my D: drive. An example of creating a JPG file is also shown commented out in the code example.

**A Note on DEVICE and USER Coordinates**

When you define the actual pixel coordinates of your drawing ‘surface’, such as (100, 200) for example, you are indicating that it is 100 pixels width by 200 pixels height. This is referred to as the DEVICE coordinates.

The USER coordinates associated with this is what was passed to the .MoveTo() and .LineTo() methods, for example, that is basically indicating ‘what percentage’ of the DEVICE coordinates are to be referred to. Therefore, 0.5 = 50% and 0.8 = 80%, for example.

So if the DEVICE coordinates are 100 x 200, then doing something like .MoveTo(0.5, 0.8) would in effect be the surface coordinates ‘x = 50, y = 160’.

**EXAMPLE USING JUST DEVICE COORDINATES TO DRAW THE SAME IMAGE.**

Private Sub Form\_Load()

Me.ScaleMode = vbPixels

Me.AutoRedraw = True

'Create a surface

Dim Srf As cCairoSurface

Set Srf = Cairo.CreateSurface(Me.ScaleWidth, Me.ScaleHeight)

'Create a context associated with the surface created above

Dim CC As cCairoContext

Set CC = Srf.CreateContext

CC.Paint 1, Cairo.CreateSolidPatternLng(vbWhite)

CC.DrawLine 0, 0, Srf.Width, Srf.Height, , 0.2 \* Srf.Height, vbBlack

CC.DrawLine Srf.Width, 0, 0, Srf.Height, , 0.2 \* Srf.Height, vbBlack

CC.Rectangle 0, 0, Srf.Width / 2, Srf.Height / 2

CC.Fill , Cairo.CreateSolidPatternLng(vbRed, 0.8)

CC.Rectangle 0, Srf.Height / 2, Srf.Width / 2, Srf.Height / 2

CC.Fill , Cairo.CreateSolidPatternLng(vbGreen, 0.6)

CC.Rectangle Srf.Width / 2, 0, Srf.Width / 2, Srf.Height / 2

CC.Fill , Cairo.CreateSolidPatternLng(vbBlue, 0.4)

Set Me.Picture = Srf.Picture

End Sub

In the above example you see that .MoveTo() and .LineTo() has been replaced with a method called .DrawLine() that will draw a line using DEVICE coordinates directly.

You may be wondering why the .Rectangle() method here is also accepting DEVICE coordinates directly when in the previous example code it was given USER coordinates.

When I replaced the first rectangle code lines (rectangle, fill, color) with the original code used in the first example using the USER coordinate values, this did not work. Why did it work in the original example but not in the one above?

**Information past this point has yet to be updated for VB6.**

**Working with Gradients**

Gradients describe a progression of colors by setting a start and stop reference location and a series of "stops" along the way. [Linear gradients](http://www.cairographics.org/manual/cairo-cairo-pattern-t.html#cairo-pattern-create-linear) are built from two points which pass through parallel lines to define the start and stop locations. [Radial gradients](http://www.cairographics.org/manual/cairo-cairo-pattern-t.html#cairo-pattern-create-radial) are also built from two points, but each has an associated radius of the circle on which to define the start and stop locations. Stops are added to the gradient with [cairo\_add\_color\_stop\_rgb()](http://www.cairographics.org/manual/cairo-cairo-pattern-t.html#cairo-pattern-add-color-stop-rgb) and [cairo\_add\_color\_stop\_rgba()](http://www.cairographics.org/manual/cairo-cairo-pattern-t.html#cairo-pattern-add-color-stop-rgba) which take a color like cairo\_set\_source\_rgb\*(), as well as an offset to indicate where it lies between the reference locations. The colors between adjacent stops are averaged over space to form a smooth blend. Finally, the behavior beyond the reference locations can be controlled with [cairo\_set\_extend()](http://www.cairographics.org/manual/cairo-cairo-pattern-t.html" \l "cairo-pattern-set-extend).

int i, j;

cairo\_pattern\_t \*radpat, \*linpat;

radpat = cairo\_pattern\_create\_radial (0.25, 0.25, 0.1, 0.5, 0.5, 0.5);

cairo\_pattern\_add\_color\_stop\_rgb (radpat, 0, 1.0, 0.8, 0.8);

cairo\_pattern\_add\_color\_stop\_rgb (radpat, 1, 0.9, 0.0, 0.0);

for (i=1; i<10; i++)

for (j=1; j<10; j++)

cairo\_rectangle (cr, i/10.0 - 0.04, j/10.0 - 0.04, 0.08, 0.08);

cairo\_set\_source (cr, radpat);

cairo\_fill (cr);

linpat = cairo\_pattern\_create\_linear (0.25, 0.35, 0.75, 0.65);

cairo\_pattern\_add\_color\_stop\_rgba (linpat, 0.00, 1, 1, 1, 0);

cairo\_pattern\_add\_color\_stop\_rgba (linpat, 0.25, 0, 1, 0, 0.5);

cairo\_pattern\_add\_color\_stop\_rgba (linpat, 0.50, 1, 1, 1, 0);

cairo\_pattern\_add\_color\_stop\_rgba (linpat, 0.75, 0, 0, 1, 0.5);

cairo\_pattern\_add\_color\_stop\_rgba (linpat, 1.00, 1, 1, 1, 0);

cairo\_rectangle (cr, 0.0, 0.0, 1, 1);

cairo\_set\_source (cr, linpat);

cairo\_fill (cr);

Images include both surfaces loaded from existing files with [cairo\_image\_surface\_create\_from\_png()](http://www.cairographics.org/manual/cairo-PNG-Support.html" \l "cairo-image-surface-create-from-png) and surfaces created from within cairo as an earlier destination. As of cairo 1.2, the easiest way to make and use an earlier destination as a source is with [cairo\_push\_group()](http://www.cairographics.org/manual/cairo-cairo-t.html" \l "cairo-push-group) and either [cairo\_pop\_group()](http://www.cairographics.org/manual/cairo-cairo-t.html" \l "cairo-pop-group) or [cairo\_pop\_group\_to\_source()](http://www.cairographics.org/manual/cairo-cairo-t.html" \l "cairo-pop-group-to-source). Use cairo\_pop\_group\_to\_source() to use it just until you select a new source, and cairo\_pop\_group() when you want to save it so you can select it over and over again with [cairo\_set\_source()](http://www.cairographics.org/manual/cairo-cairo-t.html" \l "cairo-set-source).

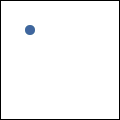
**Creating a Path**

Cairo always has an active path. If you call <cCairoContextObject>.Stroke it will draw the path with your line settings. If you call [cairo\_fill()](https://www.cairographics.org/tutorial/" \l "L3fill) it will fill the inside of the path. But as often as not, the path is empty, and both calls will result in no change to your destination. Why is it empty so often? For one, it starts that way; but more importantly after each <cCairoContextObject>.Stroke or [cairo\_fill()](https://www.cairographics.org/tutorial/" \l "L3fill) it is emptied again to let you start building your next path.

What if you want to do multiple things with the same path? For instance to draw a red rectangle with a black border, you would want to fill the rectangle path with a red source, then stroke the same path with a black source. A rectangle path is easy to create multiple times, but a lot of paths are more complex.

Cairo supports easily reusing paths by having alternate versions of its operations. Both draw the same thing, but the alternate doesn't reset the path. For stroking, alongside <cCairoContextObject>.Stroke there is [cairo\_stroke\_preserve()](http://www.cairographics.org/manual/cairo-cairo-t.html" \l "cairo-stroke-preserve); for filling, [cairo\_fill\_preserve()](http://www.cairographics.org/manual/cairo-cairo-t.html" \l "cairo-fill-preserve) joins [cairo\_fill()](http://www.cairographics.org/manual/cairo-cairo-t.html" \l "cairo-fill). Even setting the clip has a preserve variant. Apart from choosing when to preserve your path, there are only a couple common operations.

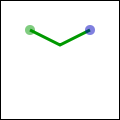
**Moving**



Cairo uses a connect-the-dots style system when creating paths. Start at 1, draw a line to 2, then 3, and so forth. When you start a path, or when you need to start a new sub-path, you want it to be like point 1: it has nothing connecting to it. For this, use [cairo\_move\_to()](http://www.cairographics.org/manual/cairo-Paths.html" \l "cairo-move-to). This sets the current reference point without making the path connect the previous point to it. There is also a relative coordinate variant, [cairo\_rel\_move\_to()](http://www.cairographics.org/manual/cairo-Paths.html" \l "cairo-rel-move-to), which sets the new reference a specified distance away from the current reference instead. After setting your first reference point, use the other path operations which both update the reference point and connect to it in some way.

cairo\_move\_to (cr, 0.25, 0.25);

**Straight Lines**

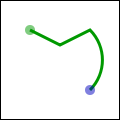


Whether with absolute coordinates [cairo\_line\_to()](http://www.cairographics.org/manual/cairo-Paths.html" \l "cairo-line-to) (extend the path from the reference to this point), or relative coordinates [cairo\_rel\_line\_to()](http://www.cairographics.org/manual/cairo-Paths.html" \l "cairo-rel-line-to) (extend the path from the reference this far in this direction), the path connection will be a straight line. The new reference point will be at the other end of the line.

cairo\_line\_to (cr, 0.5, 0.375);

cairo\_rel\_line\_to (cr, 0.25, -0.125);

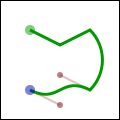
**Arcs**



Arcs are parts of the outside of a circle. Unlike straight lines, the point you directly specify is not on the path. Instead it is the center of the circle that makes up the addition to the path. Both a starting and ending point on the circle must be specified, and these points are connected either clockwise by [cairo\_arc()](http://www.cairographics.org/manual/cairo-Paths.html" \l "cairo-arc) or counter-clockwise by [cairo\_arc\_negative()](http://www.cairographics.org/manual/cairo-Paths.html" \l "cairo-arc-negative). If the previous reference point is not on this new curve, a straight line is added from it to where the arc begins. The reference point is then updated to where the arc ends. There are only absolute versions.

cairo\_arc (cr, 0.5, 0.5, 0.25 \* sqrt(2), -0.25 \* M\_PI, 0.25 \* M\_PI);

**Curves**



Curves in cairo are cubic Bézier splines. They start at the current reference point and smoothly follow the direction of two other points (without going through them) to get to a third specified point. Like lines, there are both absolute ([cairo\_curve\_to()](http://www.cairographics.org/manual/cairo-Paths.html" \l "cairo-curve-to)) and relative ([cairo\_rel\_curve\_to()](http://www.cairographics.org/manual/cairo-Paths.html" \l "cairo-rel-curve-to)) versions. Note that the relative variant specifies all points relative to the previous reference point, rather than each relative to the preceding control point of the curve.

cairo\_rel\_curve\_to (cr, -0.25, -0.125, -0.25, 0.125, -0.5, 0);

**Close the path**

[](https://www.cairographics.org/tutorial/path-close.c)

Cairo can also close the path by drawing a straight line to the beginning of the current sub-path. This straight line can be useful for the last edge of a polygon, but is not directly useful for curve-based shapes. A closed path is fundamentally different from an open path: it's one continuous path and has no start or end. A closed path has no line caps for there is no place to put them.

cairo\_close\_path (cr);

**Text**

Finally text can be turned into a path with [cairo\_text\_path()](http://www.cairographics.org/manual/cairo-Paths.html" \l "cairo-text-path). Paths created from text are like any other path, supporting stroke or fill operations. This path is placed anchored to the current reference point, so [cairo\_move\_to()](https://www.cairographics.org/tutorial/" \l "L3moving) your desired location before turning text into a path. However there are performance concerns to doing this if you are working with a lot of text; when possible you should prefer using the verb [cairo\_show\_text()](https://www.cairographics.org/tutorial/" \l "L3showtext) over [cairo\_text\_path()](https://www.cairographics.org/tutorial/" \l "L3text) and [cairo\_fill()](https://www.cairographics.org/tutorial/" \l "L3fill).

**Understanding Text**

[](https://www.cairographics.org/tutorial/textextents.c)

To use text effectively you need to know where it will go. The methods [cairo\_font\_extents()](http://www.cairographics.org/manual/cairo-text.html" \l "cairo-font-extents) and [cairo\_text\_extents()](http://www.cairographics.org/manual/cairo-text.html" \l "cairo-text-extents) get you this information. Since this diagram is hard to see so small, I suggest getting its [source](https://www.cairographics.org/tutorial/textextents.c) and bump the size up to 600. It shows the relation between the reference point (red dot); suggested next reference point (blue dot); bounding box (dashed blue lines); bearing displacement (solid blue line); and height, ascent, baseline, and descent lines (dashed green).

The reference point is always on the baseline. The descent line is below that, and reflects a rough bounding box for all characters in the font. However it is an artistic choice intended to indicate alignment rather than a true bounding box. The same is true for the ascent line above. Next above that is the height line, the artist-recommended spacing between subsequent baselines. All three of these are reported as distances from the baseline, and expected to be positive despite their differing directions.

The bearing is the displacement from the reference point to the upper-left corner of the bounding box. It is often zero or a small positive value for x displacement, but can be negative x for characters like j as shown; it's almost always a negative value for y displacement. The width and height then describe the size of the bounding box. The advance takes you to the suggested reference point for the next letter. Note that bounding boxes for subsequent blocks of text can overlap if the bearing is negative, or the advance is smaller than the width would suggest.

In addition to placement, you also need to specify a face, style, and size. Set the face and style together with [cairo\_select\_font\_face()](http://www.cairographics.org/manual/cairo-text.html" \l "cairo-select-font-face), and the size with [cairo\_set\_font\_size()](http://www.cairographics.org/manual/cairo-text.html" \l "cairo-set-font-size). If you need even finer control, try getting a [cairo\_font\_options\_t](http://www.cairographics.org/manual/cairo-cairo-font-options-t.html" \l "cairo-font-options-t) with [cairo\_get\_font\_options()](http://www.cairographics.org/manual/cairo-text.html" \l "cairo-get-font-options), tweaking it, and setting it with [cairo\_set\_font\_options()](http://www.cairographics.org/manual/cairo-text.html" \l "cairo-set-font-options).

**Working with Transforms**

Transforms have three major uses. First they allow you to set up a coordinate system that's easy to think in and work in, yet have the output be of any size. Second they allow you to make helper functions that work at or around a (0, 0) but can be applied anywhere in the output image. Thirdly they let you deform the image, turning a circular arc into an elliptical arc, etc. Transforms are a way of setting up a relation between two coordinate systems. The device-space coordinate system is tied to the surface, and cannot change. The user-space coordinate system matches that space by default, but can be changed for the above reasons. The helper functions [cairo\_user\_to\_device()](http://www.cairographics.org/manual/cairo-Transformations.html#cairo-user-to-device) and [cairo\_user\_to\_device\_distance()](http://www.cairographics.org/manual/cairo-Transformations.html#cairo-user-to-device-distance) tell you what the device-coordinates are for a user-coordinates position or distance. Correspondingly [cairo\_device\_to\_user()](http://www.cairographics.org/manual/cairo-Transformations.html#cairo-device-to-user) and [cairo\_device\_to\_user\_distance()](http://www.cairographics.org/manual/cairo-Transformations.html#cairo-device-to-user-distance) tell you user-coordinates for a device-coordinates position or distance. Remember to send positions through the non-distance variant, and relative moves or other distances through the distance variant.

I leverage all of these reasons to draw the diagrams in this document. Whether I'm drawing 120 x 120 or 600 x 600, I use [cairo\_scale()](http://www.cairographics.org/manual/cairo-Transformations.html" \l "cairo-scale) to give me a 1.0 x 1.0 workspace. To place the results along the right column, such as in the discussion of [cairo's drawing model](https://www.cairographics.org/tutorial/" \l "L1drawingmodel), I use [cairo\_translate()](http://www.cairographics.org/manual/cairo-Transformations.html" \l "cairo-translate). And to add the perspective view for the overlapping layers, I set up an arbitrary deformation with [cairo\_transform()](http://www.cairographics.org/manual/cairo-Transformations.html" \l "cairo-transform) on a [cairo\_matrix\_t](http://www.cairographics.org/manual/cairo-cairo-matrix-t.html" \l "cairo-matrix-t).

To understand your transforms, read them bottom to top, applying them to the point you're drawing. To figure out which transforms to create, think through this process in reverse. For example if I want my 1.0 x 1.0 workspace to be 100 x 100 pixels in the middle of a 120 x 120 pixel surface, I can set it up one of three ways:

1. cairo\_translate (cr, 10, 10); cairo\_scale (cr, 100, 100);
2. cairo\_scale (cr, 100, 100); cairo\_translate (cr, 0.1, 0.1);
3. cairo\_matrix\_t mat; cairo\_matrix\_init (&mat, 100, 0, 0, 100, 10, 10); cairo\_transform (cr, &mat);

Use the first when relevant because it is often the most readable; use the third when necessary to access additional control not available with the primary functions.

Be careful when trying to draw lines while under transform. Even if you set your line width while the scale factor was 1, the line width setting is always in user-coordinates and isn't modified by setting the scale. While you're operating under a scale, the width of your line is multiplied by that scale. To specify a width of a line in pixels, use [cairo\_device\_to\_user\_distance()](http://www.cairographics.org/manual/cairo-Transformations.html" \l "cairo-device-to-user-distance) to turn a (1, 1) device-space distance into, for example, a (0.01, 0.01) user-space distance. Note that if your transform deforms the image there isn't necessarily a way to specify a line with a uniform width.

**Where to Go Next**

This wraps up the tutorial. It doesn't cover all functions in cairo, so for some "advanced" lesser-used features, you'll need to look elsewhere. The code behind the examples ([layer diagrams](https://www.cairographics.org/tutorial/diagram.c), [drawing illustrations](https://www.cairographics.org/tutorial/draw.c)) uses a handful of techniques that aren't described within, so analyzing them may be a good first step. Other [examples](https://www.cairographics.org/examples/) on cairographics.org lead in different directions. As with everything, there's a large gap between knowing the rules of the tool, and being able to use it well. The final section of this document provides some ideas to help you traverse parts of the gap.

**Tips and Tricks**

In the previous sections you should have built up a firm grasp of the operations cairo uses to create images. In this section I've put together a small handful of snippets I've found particularly useful or non-obvious. I'm still new to cairo myself, so there may be other better ways to do these things. If you find a better way, or find a cool way to do something else, let me know and perhaps I can incorporate it into these tips.

**Line Width**

When you're working under a uniform scaling transform, you can't just use pixels for the width of your line. However it's easy to translate it with the help of cairo\_device\_to\_user\_distance() (assuming that the pixel width is 1):

double ux=1, uy=1;

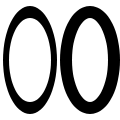
cairo\_device\_to\_user\_distance (cr, &ux, &uy);

if (ux < uy)

ux = uy;

cairo\_set\_line\_width (cr, ux);

When you're working under a deforming scale, you may wish to still have line widths that are uniform in device space. For this you should return to a uniform scale before you stroke the path. In the image, the arc on the left is stroked under a deformation, while the arc on the right is stroked under a uniform scale.

[](https://www.cairographics.org/tutorial/tips-ellipse.c)

cairo\_set\_line\_width (cr, 0.1);

cairo\_save (cr);

cairo\_scale (cr, 0.5, 1);

cairo\_arc (cr, 0.5, 0.5, 0.40, 0, 2 \* M\_PI);

cairo\_stroke (cr);

cairo\_translate (cr, 1, 0);

cairo\_arc (cr, 0.5, 0.5, 0.40, 0, 2 \* M\_PI);

cairo\_restore (cr);

cairo\_stroke (cr);

**Text Alignment**

When you try to center text letter by letter at various locations, you have to decide how you want to center it. For example the following code will actually center letters individually, leading to poor results when your letters are of different sizes. (Unlike most examples, here I assume a 26 x 1 workspace.)

F:\Downloads\2022-01-25_17-43-20.gif

cairo\_text\_extents\_t te;

char alphabet[] = "AbCdEfGhIjKlMnOpQrStUvWxYz";

char letter[2];

for (i=0; i < strlen(alphabet); i++) {

\*letter = '\0';

strncat (letter, alphabet + i, 1);

cairo\_text\_extents (cr, letter, &te);

cairo\_move\_to (cr, i + 0.5 - te.x\_bearing - te.width / 2,

0.5 - te.y\_bearing - te.height / 2);

cairo\_show\_text (cr, letter);

}

Instead the vertical centering must be based on the general size of the font, thus keeping your baseline steady. Note that the exact positioning now depends on the metrics provided by the font itself, so the results are not necessarily the same from font to font.

F:\Downloads\2022-01-25_17-45-04.gif

cairo\_font\_extents\_t fe;

cairo\_text\_extents\_t te;

char alphabet[] = "AbCdEfGhIjKlMnOpQrStUvWxYz";

char letter[2];

cairo\_font\_extents (cr, &fe);

for (i=0; i < strlen(alphabet); i++) {

\*letter = '\0';

strncat (letter, alphabet + i, 1);

cairo\_text\_extents (cr, letter, &te);

cairo\_move\_to (cr, i + 0.5 - te.x\_bearing - te.width / 2,

0.5 - fe.descent + fe.height / 2);

cairo\_show\_text (cr, letter);

}